16 Microsound

Alberto de Campo

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 most comprehensive 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. Though implementations of these variants are scattered across platforms, many of them were realized in earlier incarnations of SuperCollider.

This chapter provides a collection of detailed example implementations of many fundamental concepts of particle-based synthesis which may serve as starting points for adaptations, extensions, and further explorations by readers.

Human ears perceive sounds at different time scales quite differently; therefore, examples are provided which allow perceptual experiments with sound materials at the micro time scale.

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 of infinite amplitude but also infinitely short, so its integral is precisely 1.

In 1947 Dennis Gabor answered the question "What do we hear?" in an unusual way (Gabor, 1947): instead of illustrating quantum wave mechanics with acoustical phenomena, he did the opposite—applying a formalism from quantum physics 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. This view influenced Iannis Xenakis by 1960 (Xenakis, 1992) to consider sounds as masses of particles he called "grains" that can be shaped by mathematical means.

Sounds at the micro time scale (below, say, 100 milliseconds) became accessible for creative experiments following the general availability of computers. While much computer music models electronic devices (such as oscillators, filters, or envelope generators), a number of pioneers have created programs to generate sound involving decisions at the sample time scale. These 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 GENDYN program in 1991; see also Hoffmann (2000), Luque (2006), and N. Collins's Gendy UGens for SuperCollider).

Following concepts by Xenakis, Curtis Roads began experimenting with granular synthesis on mainframe computers in 1974; 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 has called for a change of metaphor for music composition based on his experience of making electroacoustic music (Wishart, 1994): 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 a nearly infinite differentiation in creating synthetic grains, though this is 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 (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 in a brook or in a frying pan.

Some musical instruments can be described and modeled as granular impact sounds passed through filters and resonators: guiro, 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 Matching Pursuit Wavelet Analysis has been employed to create granular representations of sound which offer new possibilities for sound visualization and transformation (Sturm et al., 2006, 2008).

16.2 Perception at the Micro Time Scale

As human listeners perceive sound events quite differently depending on the different time scales at which they occur, it is quite informative to experiment with the particularities of perception at the micro time scale. While there is a rich and interesting literature on psychoacoustics (Moore, 2004; Buser and Imbert, 1992), it tends to focus on a rather limited repertoire of sounds. For exploring new sound material, making one's own experiments (informed by psychoacoustics) can provide invaluable listening experience.

Pulses repeating at less than about 16 Hz will appear to most listeners as individual pulses, while pulses at 30 Hz fuse into continuous tones. A perceptual transition happens in between:

```
{Impulse.ar (XLine.kr(12, 48, 6, doneAction: 2)) * 0.1!2}.play; // up
{Impulse.ar (XLine.kr(48, 12, 6, doneAction: 2)) * 0.1!2}.play; // down
{Impulse.ar (MouseX.kr(12, 48, 1)) * 0.1 ! 2}.play; // mouse-controlled
```

We are quite sensitive to periodicities at different time scales; periodic pulses are perceived as pitches if they repeat 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).

Very short grains seem softer than longer ones, because loudness impression is formed over longer time windows (in psychoacoustics, this is called temporal integration). One can try comparing 2 alternating grains and adjusting their amplitudes until they seem equal.

```
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
```

```
(   // a gabor grain, gaussian-shaped envelope
SynthDef(\gabor, { |out, freq = 440, sustain = 1, pan, amp = 0.1, width = 0.25 |
    var env = LFGauss.ar(sustain, width, loop: 0, doneAction: 2);
    var son = FSinOsc.ar(freq, 0.5pi, env);
    OffsetOut.ar(out, Pan2.ar(son, pan, amp));

}, \ir ! 6).add;

    // or an approximation with a sine-shaped envelope
SynthDef(\gabor1, { |out, amp=0.1, freq=440, sustain=0.01, pan|
    var snd = FSinOsc.ar(freq);
    var env = EnvGen.ar(Env.sine(sustain, amp), doneAction: 2);
    OffsetOut.ar(out, Pan2.ar(snd * env, pan));
}, \ir ! 5).add;
)

(
Pbindef(\grain,
    \instrument, \gabor, \freq, 1000,
    \dur, 0.5, \sustain, 20/1000, \amp, 0.2
).play;
)
Pbindef(\grain, \sustain, 10/Pkey(\freq));
Pbindef(\grain, \sustain, 5/Pkey(\freq));
Pbindef(\grain, \sustain, 3/Pkey(\freq));
Pbindef(\grain, \sustain, 2/Pkey(\freq));
Pbindef(\grain, \sustain, 1/Pkey(\freq));

    // successively shorter, end
Pbindef(\grain, \sustain, Pseq((10..1)) / Pkey(\freq)).play;

    // random drift of grain duration
Pbindef(\grain, \sustain, Pbrown(1, 10, 3) / Pkey(\freq), \dur, 0.1).play
```

Figure 16.1
Short grain durations, transition from pitched to colored clicks.

```
(
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 };

p.pop
```

Figure 16.2
Perception of short silences.

```
Pbindef(\grain, \sustain, Pseq([0.001, 0.1], inf),\amp, Pseq([0.4, 0.1],
inf) );
```

With nearly rectangular envelope grains, the effect is even stronger (see example on the book Web site).

Short silences imposed on continuous sounds have different effects dependent on the 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).

When granular sounds are played almost simultaneously, the order in which they occur can be difficult to discern. Figure 16.3 plays 2 sounds with an adjustable lag between them. Lags above around 0.03 second are easily audible; shorter ones become more difficult.

In fast sequences of granular sounds, order is hard to discern, as the grains fuse into 1 sound object. But when the order changes, the new composite does sound different (see figure 16.4).

```
(
            // 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 between them
Pbindef(\hi, \lag, 0.1);
Pbindef(\hi, \lag, 0.03);
Pbindef(\hi, \lag, 0.01);
Pbindef(\hi, \lag, 0.003);

    // hi too early or too late by a fixed time - which one is first?
Pbindef(\hi, \lag, ([-1, 1].choose * 0.01).postln).play;
Pbindef(\hi, \lag, ([-1, 1].choose * 0.02).postln);

    // is it 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.

```
(
Pbindef(\grain4,
    \instrument, \percSin, \sustain, 0.03, \amp, 0.2,
    \freq, Pshuf([1000, 600, 350, 250]), // random every each time
    \dur, 0.005
).play;
                // repeat grain cluster
Tdef(\grain, { loop { Pbindef(\grain4).play; 1.wait } }).play;
)
    // fixed order
Pbindef(\grain4, \freq, Pseq([1000, 600, 350, 250].scramble));

    // different order every time
Pbindef(\grain4, \freq, Pshuf([1000, 600, 350, 250]));
```

Figure 16.4
Multiple grains fuse into one composite.

16.3 Grains and Clouds

Any sound particle shorter than about 100 ms (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 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 3 signals are plotted, as shown in figure 16.5.

```
e = Env.sine.asSignal(400).as(Array);
w = Array.fill(400, {|i| (i * 2pi/40).sin});
```

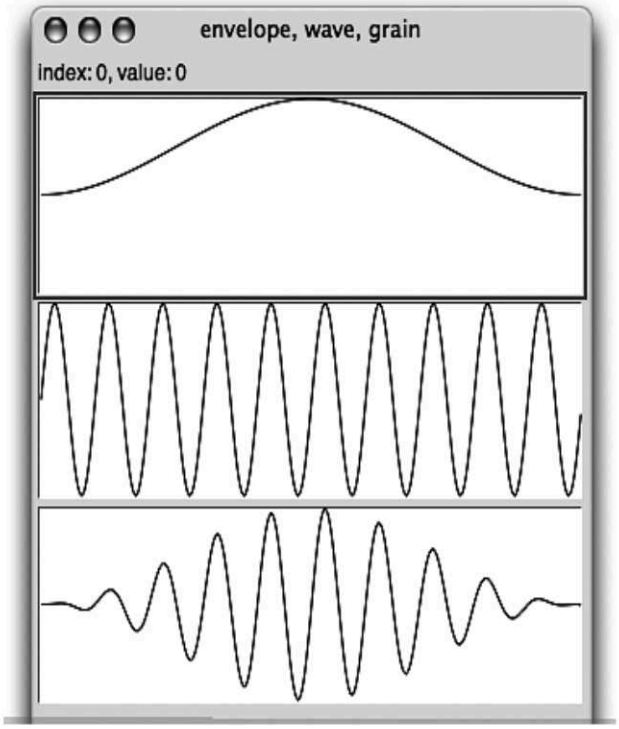

Figure 16.5
Envelope, waveform, grain.

```
g = e * w;
[e, w, g].flop.flat.plot2("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) }.plot2` or the LFGauss help file.

Assembling these elements in a `SynthDef` allows creating many variants of 1 kind of grain by varying waveform, frequency, grain duration, amplitude, and spatial position:

```
(
SynthDef(\gabor0, {|out, freq = 440, sustain = 0.02, amp = 0.2, pan|
    var env = EnvGen.ar(Env.sine(sustain, amp), doneAction: 2);
    var sound = SinOsc.ar(freq) * env;
    OffsetOut.ar(out, Pan2.ar(sound, pan))
}, \ir.dup(5)).add;
)
Synth(\gabor0);     // test with synth
Synth(\gabor0, [\freq, 1000, \sustain, 0.005, \amp, 0.1, \pan, 0.5]);
```

```
Env.sine.plot2;     // approx. gaussian
Env([0, 1, 1, 0], [0.25, 0.5, 0.25] * 0.1, \sin).test.plot2; // quasi-gaussian
Env([0, 1, 1, 0], [0.25, 0.5, 0.25] * 0.1, \lin).test.plot2; // 3 stage line
segments.
Env([0, 1, 1, 0], [0.25, 0.5, 0.25] * 0.1, \welch).test.plot2; // welch curve
interpolation
Env([1, 0.001], [0.1], \exp).test.plot2;   // expoDec (exponential decay);
Env([0.001, 1], [0.1], \exp).test.plot2;   // revExpoDec (reverse exponential decay);
Env.perc(0.01, 0.09).test.plot2;

(   // a sinc function envelope
q = q ? ();
q.makeSinc = { |q, num=1, size=400|
    dup({ |x| x = x.linlin(0, size-1, -pi, pi) * num; sin(x) / x }, size);
};
a = q.makeSinc(6);
a.plot(bounds: Rect(0,0,409,200), minval: -1, maxval: 1);
)
```

Figure 16.6
Making different envelope shapes.

```
(instrument: \gabor0).play; // test with event
(instrument: \gabor0, sustain: 0.001, freq: 2500, amp: 0.05, pan: -0.5).
play;
Synth.grain(\gabor0, [\freq, 2000, \sustain, 0.003]) // higher efficiency,
as no NodeID is kept

s.sendMsg("s_new", \gabor0, -1, 0, 0, \freq, 2000, \sustain, 0.003); //
even more efficient, as no Synth object is created.
```

This example demonstrates recommended practices for granular synthesis. As grains may have extremely short durations, audio-rate envelopes are preferred. Timing between grains should be as accurate as possible, as the ear is very sensitive to microrhythmic variations; thus one uses `OffsetOut` to start the grain's synthesis process with single-sample accuracy. Since grain synthesis parameters typically do not change while the grain plays, efficiency can be optimized with `\ir` arguments. The 2 final lines show possible trade-offs of CPU load versus programming effort. `Synth.grain` creates nodes without nodeIDs (reducing overhead), and `s.sendMsg` constructs the same even more efficiently with nodeID –1: it does not create any language side Synth object, but directly messages the Server. Finally, adding synthdefs also makes them available for use in patterns. The tests shown play single grains both as synths and as events, using every parameter at least once to verify that they work correctly.

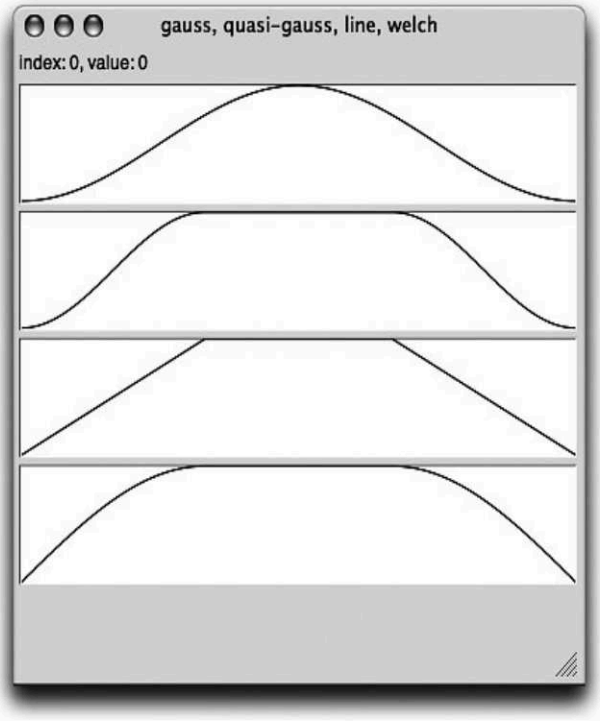

Figure 16.7
Envelopes: Gaussian, quasi-Gaussian, linear, welch.

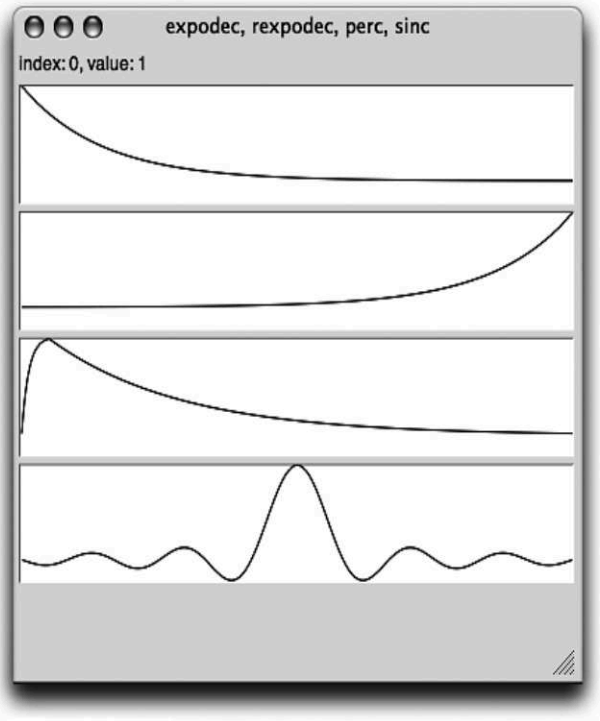

Figure 16.8
Envelopes: expodec, rexpodec, percussive, sinc-function.

```
(   // 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 middle.
SynthDef(\gabWide, { |out, amp=0.1, freq=440, sustain=0.01, pan, width=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 ! 5).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;
```

Figure 16.9
SynthDefs with different envelopes.

```
            // 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
(continued)

SC3 allows for quick testing of envelope variants; below, we create a number of common envelopes which can have quite 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; *quasi-Gaussian* 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.)

With synthdefs using these envelopes (see figure 16.9), we can experiment with the fundamental grain parameters: waveform frequency, envelope shape, and grain duration.

In order to access all parameters of the granular stream while it is playing, figure 16.10 uses the `Pbindef` class (see its Help file).

Parameter changes can have side effects of interest. For example, the `\rexpodec` synthdef's (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 (for-

```
    // figure 16.10 - changing grain duration, frequency, envelope
(
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, \percSinRev], inf));
```

Figure 16.10
Changing grain duration, frequency, envelope.

```
(   // 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-changing 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(0.01,
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 drift
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  })
);
)
```

Figure 16.11
Different control strategies applied to density.

```
    // different freq movement, different timing
Pbindef(\grain0, \freq, Pbrown(48.0, 96.0, 12.0).midicps);

(   // duration depends on freq, 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
(continued)

ward) \expodec may employ this transient for attack shaping: by adjusting the oscillator's initial phase, different attack colors can be articulated (see, e.g., the book Web site).

16.3.2 Textures, Masses, Clouds

When shifting 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; texture is a rather flexible term used by, among others, Trevor Wishart (Wishart, 1994); Edgard Varèse often spoke of masses and *volumina* of sound; the cloud metaphor suggests interesting vocabulary for imagining clouds of sound particles, inspired by the rich morphologies of clouds in 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. *Quasi-synchronous* 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: 125–129).

These strategies can be applied to any control parameter. Figure 16.12 shows different combinations of control strategies and grain and cloud parameters described so far.

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. Here, the discussion focuses on the synthesis control techniques and the aesthetic aspects; coding style aspects are covered in chapter 5. (For the code, see *CloudGenMiniFull.scd* on the book Web site.) *CloudGenMini* provides a collection of synthdefs for different sound particle flavors, a `Tdef` that creates the grain cloud based on current parameter settings, crossfading between stored settings, and a GUI for playing it. This allows creating clouds based on tendency masks, which was a central feature of *CloudGenerator*.

The `Tdef(\cloud0)` plays a loop which creates values for each next grain by random choice 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 these by making manual changes, one may spend much time in relatively uninteresting areas. One common heuristic that addresses this is to create random ranges which may assist in finding more interesting zones. *CloudGenMini* can create random ranges for all synthesis and cloud parameters, within global maximum settings, and can switch or interpolate between 8 stored range settings.

CloudGenerator allowed specifying a total cloud duration, start and end values for high and low *band limit* (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 crossfading 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

```
(
Pbindef(\grain0).clear;
Pbindef(\grain0,
    \instrument, \expodec,
    \freq, 200,
    \sustain, 0.05, \dur, 0.07
).play;
)
    // time-varying freq with envelope pattern
Pbindef(\grain0, \freq, Pn(Penv([200, 1200], [10], \exp), inf));
    // random freq
Pbindef(\grain0, \freq, 400 * Pwhite(-24.0, 24).midiratio);
    // timechanging with random variation
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)); // rhythmic random

    // amplitude - randomized
Pbindef(\grain0, \amp, Pwhite(0.01, 0.2)); // linear
Pbindef(\grain0, \amp, Pwhite(-50, -14).dbamp); // exponential - more depth
Pbindef(\grain0, \dur, 0.025 * Prand([0, 1, 1, 2, 4], inf)); // could 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 freq
Pbindef(\grain0, \sustain, Pkey(\freq).reciprocal * 20).play;
```

Figure 16.12
Control strategies applied to different parameters.

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 1 synthesis process. However, SC3 also has a selection of UGens that implement granular synthesis entirely on the sound server. Though one can obtain similar sounds using either approach, it is interesting to experimentally learn how parameter control by UGens suggests different solutions and thus leads 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 the UGens `GrainSin`, `GrainFM`, `GrainBuf`, `GrainIn`, and `Warp1` became part of the SC3 distribution. Third-party UGen libraries (as in sc3-plugins) and Quarks are worth checking for more granular synthesis variants and options.

`GrainSin` creates a stream of grains with a sine waveform and a Hanning-shaped envelope; grains are triggered by a control signal's transition from negative to positive. Below is a simple conversion of the `GrainSin` Help file example to JIT style (see chapter 7). Briefly, a `ProxySpace` is an environment for NodeProxies (placeholders for synthesis processes). NodeProxies can be changed and reconfigured very flexibly while running, making them ideally suited for fluid exploration of synthesis variants.

```
p = ProxySpace.push;
(
~grain.play;
~grain = {arg envbuf = -1, density = 10, graindur = 0.1, amp = 0.2;
    var pan, env, freqdev;
    var trig = Impulse.kr(density);
    pan = MouseX.kr(-1, 1);       // use mouse-x for panning
    // use WhiteNoise and mouse Y to control deviation from center
    freqdev = WhiteNoise.kr(MouseY.kr(400, 0));
    GrainSin.ar(2, trig, graindur, 440 + freqdev, pan, envbuf) * amp
};
)
```

`GrainSin` allows for custom grain envelopes, which must be uploaded as buffers on the server. In the next example, an envelope is converted to a `Signal` and is sent to a buffer, and the `~grain` proxy is set to use that buffer number. A `bufnum` of –1 sets it to the default envelope.

```
q = q ? ();     // make a dictionary to keep things around
q.envs = ();    // space for some envelopes
```

```
q.bufs = ();    // and some buffers
                // make an envelope and send it to a buffer
q.envs.perc1 = Env([0, 1, 0], [0.1, 0.9], -4);
q.bufs.perc1 = Buffer.sendCollection(s, q.envs.perc1.discretize, 1);
~grain.set(\envbuf, -1); // switch to built-in envelope
~grain.set(\envbuf, q.bufs.perc1.bufnum); // or customized
```

Besides changing the parameter controls of the proxy 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 distributed
~grdensity = {2 ** LFNoise0.kr(1).range(0, 6)};
    // map to density control
~grain.map(\density, ~grdensity);
```

At this point, exploration becomes more enjoyable with a `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. Below 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.

```
~grain = {arg envbuf = -1, density = 10, graindur = 0.1, modfreq = 200;
    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, moddepth, pan,
    envbuf) * 0.2
};
```

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 crossfaded. Figure 16.13 is such a rewrite, where all parameters can be changed freely between fixed values and synthesis functions, line by line, in any order.

```
    // figure 16.13   - GrainFM with individual control proxies
p = ProxySpace.push;

(
~trig = { |dens=10| Impulse.kr(dens) };
~freq = { MouseX.kr(100, 2000, 1) * LFNoise1.kr(1).range(0.25, 1.75)
};
~moddepth = { LFNoise1.kr(20).range(1, 10) };
~modfreq = 200;
~graindur = 0.1;

~grain = { arg envbuf = -1;
    GrainFM.ar(2, ~trig.kr, ~graindur.kr,
        ~freq.kr, ~modfreq.kr, ~moddepth.kr,
        pan: WhiteNoise.kr, envbufnum: envbuf) * 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
density
~freq = { LFNoise0.kr(0.3).range(200, 800) };
~moddepth = 3;  // fixed depth
~graindur = { LFNoise0.kr.range(0.01, 0.1) };
```

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 constantly moving the file read position (i.e., where in the sound file to take the next grain waveform from) and by varying 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.

`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.

```
b = Buffer.read(s, "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.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) * 0.2
};
~grain.play;
)

    // experiment with control proxies
~trig = { |dens=20| Impulse.kr(dens) };
~rate = { LFNoise1.kr.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;
```

Figure 16.14
GrainBuf with control proxies.

When comparing UGen-based and individual-grain particle syntheses, experimenting with UGens as controls allows for very interesting behavior—even much patternlike behavior can be realized with `Demand` UGens (see *Demand*.Help file). On the other hand, one cannot write one's own special grain flavors with server-side granular synthesis. Unless one is fluent enough in C++ to implement new UGens (see chapter 25), one is limited to the synthesis processes provided as UGens.

16.5 Exploring Granular Synthesis Flavors

The most flexible starting point for creating one's own microsound flavors is considering that grains can contain any waveform. Reviewing the synthdefs given in figure 16.9, all one needs to change is the sound source itself. As an exercise, we could

replicate GrainFM and GrainBuf as synthdefs; `\grainFM0` adds 3 parameters and changes the sound source to an FM pair, while `\grainFM1` uses a buffer envelope with Osc1, a pseudo-UGen class written for this chapter and available on the book Web site.

```
SynthDef(\grainFM1, {|out, envbuf, carfreq = 440, modfreq = 200, moddepth
= 1,
    sustain = 0.02, amp = 0.2, pan|
    var env = Osc1.ar(envbuf, sustain, doneAction: 2);
    var sound = SinOsc.ar(carfreq, SinOsc.ar(modfreq) * moddepth) * env;
    OffsetOut.ar(out, Pan2.ar(sound, pan, amp))
}, \ir.dup(8)).add;
```

`GrainBuf` can be re-created with `PlayBuf`. Both synthdefs can be played with patterns or tasks from sclang (examples are on the book Web site).

```
SynthDef(\grainBuf1, {|out, envbuf, wavebuf, filepos, rate = 1, sustain =
0.02, amp = 0.2, pan|
    var env = Osc1.ar(envbuf, sustain, doneAction: 2);
    var sound = PlayBuf.ar(1, wavebuf,
        rate * BufRateScale.ir(wavebuf), 1,
        startPos: BufFrames.ir(wavebuf) * filepos)
        * env;
    OffsetOut.ar(out, Pan2.ar(sound, pan, amp))
}, \ir.dup(8)).add;
```

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).

One possibility for organizing glissando structures is magnetization patterns (Roads, 2001: 121–125). See the example on the book Web site.

Pulsar Synthesis is named after pulsars, spinning neutron stars discovered in 1967 that emit electromagnetic pulses in the range of 0.25 Hz to 642 Hz. 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

```
(
b = Buffer.read(s, "sounds/a11wlk01-44_1.aiff");

SynthDef("glisson",
    { arg out = 0, envbuf, freq=800, freq2=1200, sustain=0.001,
amp=0.2, pan = 0.0;
        var env = Osc1.ar(envbuf, sustain, 2);
        var freqenv = XLine.ar(freq, freq2, sustain);
        OffsetOut.ar(out,
            Pan2.ar(SinOsc.ar(freqenv) * env, 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,
             \envbuf, b.bufnum
        ]);
        (3 / (i + 10)).wait;
    });
}).play;
)
```

Figure 16.15
Glisson synthesis.

over a given time; this concept has been expanded in the *PulsarGenerator* program (written in SC2 by the author and Curtis Roads).

The 2 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, 20 pulses are emitted per second; at a `formfreq` of 100 Hz, every pulse is scaled to 0.01 second duration, so within 0.05 second of 1 pulsar period, the *duty cycle* where signal is present is 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.

```
    // figure 16.16 - Pulsar basics - make a set of waveform and control tables
(
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 }.dup(7));

            // random tables for the control parameters:
q.curr.tab.fund = 200 ** Env({1.0.rand}!8, {1.0.rand}!7, \sin).discretize.as(Array);
q.curr.tab.form = 500 ** ( 0.5 + Env({rrand(0.0, 1.0)}!8, {1.0.rand}!7,
\sin).discretize.as(Array));
q.curr.tab.amp = 0.2.dup(1024);
q.curr.tab.pan = Signal.sineFill(1024, { 1.0.rand }.dup(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.plot2("a pulsaret");
```

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

Figure 16.17 realizes 1 pulsar train with a `GrainBuf`, initially with fixed parameter values. Changing the parameters 1 at a time and crossfading between them demonstrates the effect of movements of `formfreq` and `fundfreq`. Finally, replacing the controls with looping tables completes a minimal pulsar synthesis program.

Figure 16.18 shows the sending of 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.

PulsarGenerator realized several aspects of advanced pulsar synthesis: 3 parallel pulsar trains (sharing `fundfreq` but with independent `formfreq`, `amp`, and `pan` controls) are being driven from separate control tables. One can switch or crossfade between sets of tables. Both table sets and banks of table sets can be saved to disk.

Pulsar masking was implemented in 2 forms: *burst ratio* specified how many pulses to play and how many to mute; (e.g., 3:2 would play this sequence of pulses: 1, 1, 1, 0, 0, 1, 1, 1, 0, 0). This allowed generating subharmonics of the fundamental frequency. Alternatively, stochastic pulse masking was controlled from a table with values between 1.0 (play every pulse) and 0.0 (mute every pulse); 0.5 meant playing each pulse with a 50% chance, which could create interesting intermittency. In SC3,

```
(
p = ProxySpace.push;

        // 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], envbuf);
};
~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 nodeproxies using GrainBuf.

```
q.bufs.pulsaret.sendCollection(Array.linrand(1024, -1.0, 1.0)); // noise burst
q.bufs.pulsaret.read("sounds/a11wlk01.wav", 44100 * 1.5);       // sample
q.bufs.pulsaret.sendCollection(Pbrown(-1.0, 1.0, 0.2).asStream.nextN(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).discretize.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.rand}!7,
\sin).discretize.as(Array));
q.bufs.form.sendCollection(q.curr.tab.form);
```

Figure 16.18
Making new tables and sending them to buffers.

burst ratio could be reimplemented by multiplying the trigger signal with demand UGens to provide a sequence, and random masking with `CoinGate`.

Pulsar synthesis can also be implemented with client-side control, where one can choose to control parameters from patterns such as envelope segment players or from tables (see the examples on the book Web site). 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 and Florian Hecker realized a number of pieces with material generated by pulsar synthesis. Pulsars can be used particularly well as exciter signals for filters or as input material for convolution processes (Roads, 2001: 147–154). Tommi Keränen has implemented an SC3 version of PulsarGenerator with a slightly different feature set, which however, has not yet been officially released.

16.6 Sound Files and Microsound

Sound files are a great 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 2 classic uses of granular synthesis in this context. However, both writing more complex Synthdefs for granulating sound files (e.g., with filtering) and analyzing sound file waveforms to use them as a source for microstructure provide further areas to explore. Below we present examples for both: constant-Q granulation and wavesets.

16.6.1 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 for physics in 1971 for the invention of holography) built a "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 where in the recording the grain read begins and how fast the signal is read, time and pitch of the original can be changed.

Figure 16.19 provides a setup for experimenting. `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 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.

Tuning pitch-time changes to sound lifelike can be difficult, and many commercial applications spend much effort on it. Bringing forward details in recordings in non-naturalistic ways may be a more rewarding use for writing one's own pitch-time manipulating instruments.

16.6.2 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 16.20 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 synthesis after ring time is over.

Parameter tests (on the book Web site) demonstrate accessing different file regions and varying exciter grain duration; 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.

Figure 16.21 demonstrates creating a stream of constant-Q grains with a `Pbindef` pattern, which allows for changing all parameter patterns while playing.

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

```
p = ProxySpace.push(s.boot);
b = Buffer.read(s, "sounds/a11wlk01-44_1.aiff");
(
~timepitch = {arg sndbuf, 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(sndbuf)).range(0, 1)
        + LFNoise0.kr(grainRate, posRd);

    GrainBuf.ar(2, Impulse.kr(grainRate), graindur, sndbuf, pitchrate,
            position, 4, 0, -1)
};
~timepitch.set(\sndbuf, b.bufnum);
~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, \overlap, 4, \posSpeed,
1, \posRd, 0);

    // four times as long: tweak pitchRd and posJitter to reduce artifacts
~timepitch.set(\pitchRatio, 1, \pitchRd, 0, \grainRate, 20, \overlap, 4, \posSpeed,
0.25, \posRd, 0);

    // random read position, random pitch
~timepitch.set(\pitchRatio, 1, \pitchRd, 0.5, \grainRate, 20, \overlap, 4,
\posSpeed, 0.25, \posRd, 0.5);
```

Figure 16.19
A nodeproxy for time-pitch changing.

```
b = Buffer.read(s, "sounds/a11wlk01-44_1.aiff");
(
SynthDef(\constQ, { |out, bufnum=0, amp=0.5, pan, centerPos=0.5, sustain=0.1,
    rate=1, freq=400, rq=0.3|

    var ringtime = (2.4 / (freq * rq) * 0.66).min(0.5); // estimated
    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(bufnum),
        1) * envSig;
    var filtered = BPF.ar( grain, freq, rq, ampcomp );

    OffsetOut.ar(out, Pan2.ar(filtered, pan, cutoffEnv))
}, \ir.dup(8)).add;
)

Synth(\constQ, [\bufnum, b, \freq, exprand(100, 10000), \rq, exprand(0.01, 0.1),
\sustain, 0.01]);
```

Figure 16.20
A constant-Q Synthdef.

shape the sounds created. Many other transformation processes can be applied to each grain, with individual parameters creating finely articulated textures.

16.6.3 Wavesets

Trevor Wishart introduced the *waveset* concept in *Audible Design* (1994), implemented it in the CDP framework as transformation tools, and employed it in several compositions (e.g., *Tongues of Fire,* 1994). Though 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 1 zero-crossing of a signal to the third, so in a sine wave, it corresponds to the sine wave's period. In aperiodic signals (such as sound files) the length and shape of waveform segments vary widely. The `Wavesets` class analyzes a sound file into wavesets, maintaining zero-crossings, lengths, amplitudes, and other values of each waveset (see the *Wavesets* Help file).

```
(
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 16.21
A stream of constant-Q grains.

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. Figure 16.22 creates a waveset from a sound file and shows accessing its internal data. It also displays accessing and plotting individual wavesets or groups of wavesets. Figures 16.23 and 16.24 show a single waveset and a waveset group, respectively.

Figure 16.25 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.

Table 16.1
Overview of Waveset Transforms

Transposition	e.g., 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 2 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 1 source with waveset forms from another
Interleaving	take alternating wavesets or groups from 2 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 crossfading repetitions
Time shrinking	keep every nth waveset or group

In order to show Wishart's transforms, we use a `Pbindef`, so we can replace patterns or fixed values in it while it is running. It reconstructs part of the sound file waveset by waveset (see figure 16.26.)

Now we can introduce waveset transposition, reversal, time stretching, omission, and shuffling in 1-line examples. (See figure 16.27.)

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 2 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 16.28 also considers the amplitudes for each waveset: since the substitute signal is full volume, scaling it to the original waveset's volume keeps the dynamic contour intact. Finally, different substitute waveforms can have quite different effects.

```
w = Wavesets.from("sounds/a11wlk01.wav");

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 waveset
w.minima;       // index of negative minimum value in every waveset

w.fracXings;    // fractional zerocrossing points
w.fracLengths;  // and lengths: allows more precise looping.

w.lengths.plot; // show distribution of lengths
w.amps.plot;

    // get data for a single waveset: frameIndex, length (in frames), dur
w.frameFor(140, 1);
w.ampFor(140, 1);       // peak amplitude of that waveset or group

    // extract waveset by hand
w.signal.copyRange(w.xings[150], w.xings[151]).plot("waveset 150");
w.plot(140, 1); // convenience plotting
w.plot(1510, 1);

    // plot a group of 5 adjacent wavesets
w.plot(1510, 5)
```

Figure 16.22
A Wavesets object.

Waveset averaging can be realized by adapting this example to play *n* wavesets simultaneously at a time, 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 on the book Web site the 2 wavesets are synchronized: the sound begins with *waveset1* at original speed and *waveset2* scaled to the same loop duration, but silent. As the amplitude crossfades from *waveset1* to *waveset 2*, so does the loop speed, so that the interpolation ends with only *waveset 2* at its original speed. The example can be extended to plays a seamless stream of parameterizable interpolations.

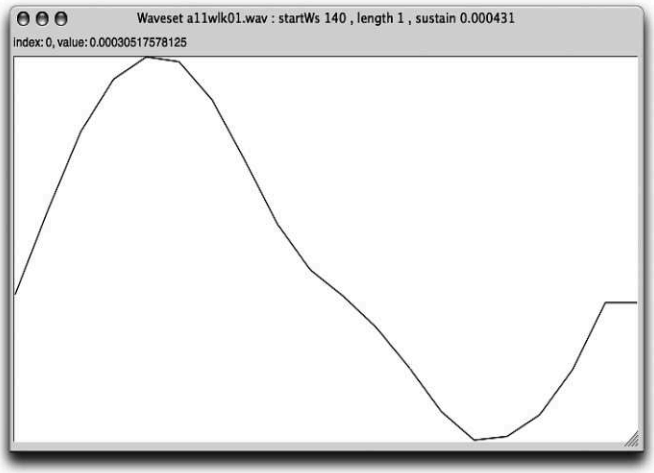

Figure 16.23
A single waveset plotted.

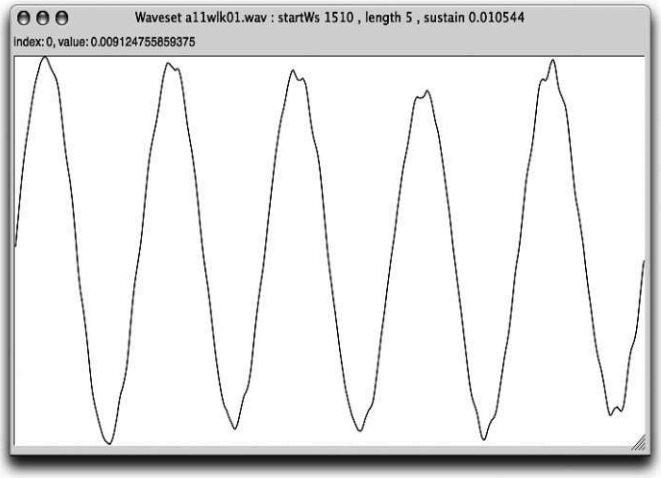

Figure 16.24
A waveset group of size 5.

```
(
    // A wavesets loads the file into a buffer by default.
        b = w.buffer;
    // Wavesets.prepareSynthDefs loads this synthdef:
        SynthDef(\wvst0, { arg out = 0, buf = 0, start = 0, length =
441, playRate = 1, sustain = 1, amp=0.2, pan;
            var phasor = Phasor.ar(0, BufRateScale.ir(buf) *
playRate, 0, length) + start;
            var env = EnvGen.ar(Env([amp, amp, 0], [sustain, 0]),
doneAction: 2);
            var snd = BufRd.ar(1, buf, phasor) * env;

            OffsetOut.ar(out, Pan2.ar(snd, pan));
        }, \ir.dup(8)).add;
)

// play from frame 0 to 440, looped for 0.1 secs, so ca 10 repeats.
(instrument: \wvst0, bufnum: b.bufnum, start: 0, length: 440, amp: 1,
sustain: 0.1).play;

    // get data from waveset
(
var start, length, sustain, repeats = 20;
#start, length, sustain = w.frameFor(150, 5);

(   instrument: \wvst0, bufnum: b.bufnum, amp: 1,
    start: start, length: length, sustain: sustain * repeats
).play;
)

    // or even simpler:
w.eventFor(startWs: 150, numWs: 5, repeats: 20, playRate:
1).put(\amp, 0.5).play;
```

Figure 16.25
Playing Wavesets from buffers.

```
    // by default, this pattern reconstructs a soundfile segment as is.
(
Pbindef(\ws1).clear;
Pbindef(\ws1,
    \instrument, \wvst0,
    \startWs, Pn(Pseries(0, 1, 3000), 1),
    \numWs, 1,
    \playRate, 1,
    \bufnum, b.bufnum,
    \repeats, 1,
    \amp, 0.4,
    [\start, \length, \sustain], Pfunc({ |ev|
        var start, length, wsDur;

        #start, length, wsDur = w.frameFor(ev[\startWs], ev[\numWs]);
        [start, length, wsDur * ev[\repeats] / ev[\playRate].abs]
    }),
    \dur, Pkey(\sustain)
).play;
)
```

Figure 16.26
A pattern to play Wavesets.

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 16.29 begins with a fixed grain repeat rate and starting waveset, later replaced with different values and streams for generating values.

In figure 16.29, 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.

A wide range of possibilities opens here: one can create special orders of the wavesets based, for example, on their lengths or amplitudes; one can filter all waveset indices by some criterion (e.g., keep only very soft ones (see figure 16.30)).

For just 1 example for modifying parameters based on waveset information, see figure 16.31. 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.

```
    // waveset transposition: every second waveset, half speed
Pbindef(\ws1, \playRate, 0.5, \startWs, Pn(Pseries(0, 2, 500), 1)).play;

    // reverse every single waveset
Pbindef(\ws1, \playRate, -1, \startWs, Pn(Pseries(0, 1, 1000), 1)).play;
    // reverse every 2 wavesets
Pbindef(\ws1, \numWs, 2, \playRate, -1, \startWs, Pn(Pseries(0, 2, 1000), 1)).play;
    // reverse every 20 wavesets
Pbindef(\ws1, \numWs, 20, \playRate, -1, \startWs, Pn(Pseries(0, 20, 1000),
1)).play;
    // restore
Pbindef(\ws1, \numWs, 1, \playRate, 1, \startWs, Pn(Pseries(0, 1, 1000), 1)).play;

    // time stretching
Pbindef(\ws1, \playRate, 1, \repeats, 2).play;
Pbindef(\ws1, \playRate, 1, \repeats, 4).play;
Pbindef(\ws1, \playRate, 1, \repeats, 6).play;
Pbindef(\ws1, \repeats, 1).play;    // restore

    // waveset omission: drop every second
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, \) }) ).play; // drop
randomly
Pbindef(\ws1, \numWs, 1, \freq, 1, \startWs, Pn(Pseries(0, 1, 1000)) ).play; //
restore

    // waveset shuffling (randomize waveset order +- 5, 25, 125)
Pbindef(\ws1, \startWs, Pn(Pseries(0, 1, 1000), 1) + Pfunc({ 5.rand2 })).play;
Pbindef(\ws1, \startWs, Pn(Pseries(0, 1, 1000), 1) + Pfunc({ 25.rand2 })).play;
Pbindef(\ws1, \startWs, Pn(Pseries(0, 1, 1000), 1) + Pfunc({ 125.rand2 })).play;
```

Figure 16.27
Some of Trevor Wishart's transforms.

```
    // the waveform to substitute
c = Buffer.alloc(s, 512); c.sendCollection(Signal.sineFill(512, [1]));
(
Pbindef(\ws1).clear;
Pbindef(\ws1,
    \instrument, \wvst0,
    \startWs, Pn(Pseries(0, 1, 1000), 5),
    \numWs, 1, \playRate, 1,
    \buf, c.bufnum, // sine wave
    \repeats, 1,
    \amp, 1,
    [\start, \length, \sustain], Pfunc({ |ev|
        var start, length, wsDur, origRate;
        origRate = ev[\playRate];

            // get orig waveset specs
        #start, length, wsDur = w.frameFor(ev[\startWs], ev[\numWs]);

            // adjust playrate for different length of substituted wave
        ev[\playRate] = origRate * (512 / length);

            // get amplitude from waveset, to scale full volume sine wave
        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, \playRate, 1, \repeats, 2).play;
Pbindef(\ws1, \playRate, 1, \repeats, 6).play;
Pbindef(\ws1).stop;

    // different waveforms
c.sendCollection(Signal.sineFill(512, 1/(1..4).squared.scramble));
c.sendCollection(Signal.rand(512, -1.0, 1.0));
c.sendCollection(Signal.sineFill(512, [1]));

c.plot;
```

Figure 16.28
Waveset substitution.

```
    // very simple first pass, fixed repeat time
(
Tdef(\ws1).set(\startWs, 400);
Tdef(\ws1).set(\numWs, 5);
Tdef(\ws1).set(\repeats, 5);

Tdef(\ws1, { |ev|
    var startFrame, length, wsSustain;

    loop {
        #startFrame, length, wsSustain = w.frameFor(ev.startWs.next,
ev.numWs);

        (instrument: \wvst0, bufnum: b.bufnum, amp: 1,
            start: startFrame, length:  length,
            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).asStream);
```

Figure 16.29
Wavesets played with a Tdef.

16.7 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 changing directions quite flexibly. To give just 1 example, the pattern library offers many ways to create intricately detailed structures that may lead to fascinating microsound textures. Independent of one's aesthetic preferences, and of pre-

```
(
Tdef(\ws1).set(\gap, 3);
Tdef(\ws1, { |ev|
    var startFrame, length, wsSustain, reps;

    loop {
        reps = ev.repeats.next;

        #startFrame, length, wsSustain =
            w.frameFor(ev.startWs.next, ev.numWs.next);


        (instrument: \wvst0, bufnum: b.bufnum, amp: 1,
            start: startFrame, length:  length,
            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 16.30
Waittime derived from waveset duration with a gap added.

```
(
Tdef(\ws1).set(\startWs, Pn(Pseries(0, 5, 400) + 500, inf).asStream);

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, length, wsSustain, reps, numWs, len2Avg;
    var squeezer, playRate;
    loop {
        reps = ev.repeats.next;
        numWs = ev.numWs.next;

        #startFrame, length, wsSustain =
            w.frameFor(ev.startWs.next, numWs);

        len2Avg = length / numWs / w.avgLength;
        squeezer = len2Avg ** ev.pitchContour.next;
        wsSustain = wsSustain / squeezer;
        playRate = 1 * squeezer;

        if (ev.keepCoin.next.coin) {
            (instrument: \wvst0, bufnum: b.bufnum, amp: 1,
                start: startFrame, length:  length,
                sustain: wsSustain * reps,
                playRate: playRate,
                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
```

Figure 16.31
Wavesets with pitch contour and dropout rate.

```
        // waveset overtone singing - all equal length
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])).asStream).play;

    // add a pitch contour envelope
Tdef(\ws1).set(\pitchContour, Pn(Penv([0, 2, 0], [21,
13])).asStream);
```

Figure 16.31
(continued)

ferred methodologies for creating music, there are plenty of possibilities for further exploration.

References

Buser, P., and M. Imbert. 1992. *Audition*. Cambridge, MA: MIT Press.

Gabor, D. 1947. "Acoustical Quanta and the Theory of Hearing." *Nature,* 159(4044): 591–594.

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.

Moore, B. C. J. 2004. *An Introduction to the Psychology of Hearing,* 5th ed. London: Academic 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.

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., J. J. Shynk, L. Daudet, and C. Roads. 2008. "Dark Energy in Sparse Atomic Estimations." *IEEE Transactions on Audio, Speech & Language Processing*, 16(3): 671–676.

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. [1971] 1992. *Formalized Music: Thought and Mathematics in Music*, rev. and enl. ed. Hillsdale, NY: Pendragon Press.