22 Microsound (2007, Alberto de Campo)

Since the pioneering work of Denis Gabor and Iannis Xenakis, the idea of composing music and sound from minute particles has merged into the mainstream of electronic music; many commercial software synthesis programs now offer granular synthesis modules, a media artist group calls itself Granular Synthesis, and a mailing list goes by the name of microsound.org.

The most thorough book on the topic, Microsound (Roads 2001), covers historical, aesthetic, and technical considerations, and provides an extensive taxonomy of variants of particle-based sound synthesis and transformation. While 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 explorations by readers.

For perceptual reasons, Microsound is fundamentally different from standard musical practice; therefore code examples which allow for some relevant experiments in auditory perception are also provided.

1 Points of Departure

Imagine a sine wave that has begun 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 answers the question "What do we hear?" in an unusual way (Gabor 1947): instead of illustrating quantum wave mechanics with acoustical phenomena, he does the opposite ó applying a formalism from quantum physics to auditory perception, he obtains 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 introduces 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 has influenced Iannis Xenakis to consider sounds as masses of particles he called 'grains' that can be shaped in mathematical terms very early on (Xenakis 1960).

Sounds at the micro-time scale (below say 100 msec) only became accessible for creative experiments by means of computers. While much computer music models electronic devices (such as oscillators, filters, envelope generators), some pioneers have created programs to generate sound by decisions on the sample time scale: Herbert Brün's SAWDUST (1976), G.M. Koenig's SSP (early 1970s), and Iannis Xenakis's Stochastic Synthesis (described first in Xenakis 1972, realized as the GENDY program 1991, see also the Gendy UGens by N. Collins).

Trevor Wishart has called for a change of metaphor for music composition based on experience made in electronic music (Wishart 1994): Rather than architecture (of pitch/time/parameter constructions), chemistry or alchemy can provide models for the possibilities of infinite malleability of sound materials in computer music. This extends to the microtime scale, as infinite differentiation of creating synthetic grains is technically possible.

Many physical sounds can be described as granular structures: Dolphins communicate by clicking sounds; many insects produce sounds, e.g. crickets make friction sounds with a rasping action (pulses), filtered by mechanical resonance of parts of their exoskeletons such as their legs; bats echolocate obstacles and possible prey in their environment by emitting short ultrasound bursts and listening to the reflections coming back.

Many sounds that involve a multitude of similar objects interacting will induce global percepts with statistical properties: rustling leaves produce myriads of single short sounds, as do pebbles when waves recede from the shore; the film sound staple of steps on gravel can be seen 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 going through filters and resonators: guiro, rainstick, rattles, maracas; fast tone repetitions on many instruments, fluttertounge effects on wind instruments; as well as any instrument that can be played with drum rolls.

1.1 Perception at the Micro-time Scale

Human ears perceive sounds at different time scales quite differently, and it is quite informative to experiment with these changes of perceptual modes.

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

```
// example 22.1 - pulses, transition from rhythm to pitch
{ 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. How often is enough? If one creates tones with very short durations, on the order of 10 waveform repetitions before they end, one can study how the percept of a pitched tone goes away and becomes more like shades of sound color of a click.

```
// example 22.2 - short grain durations - pitch to colored click
(
    // a gabor (approx. gaussian-shaped) grain
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).store;
)
(
Pbindef(\grain,
    \instrument, \gabor1,
    \dur, 0.5,
    \freq, 1000,
    \sustain, 20/1000
).play;
)
Pbindef(\grain, \sustain, 10/Pkey(\freq));
Pbindef(\grain, \sustain, 7/Pkey(\freq));
Pbindef(\grain, \sustain, 5/Pkey(\freq));
Pbindef(\grain, \sustain, 3/Pkey(\freq));
Pbindef(\grain, \sustain, 2/Pkey(\freq));
Pbindef(\grain, \sustain, 0.1/Pkey(\freq));
```

```
    // successively shorter, end
Pbindef(\grain, \sustain, Pseq((10..1), inf)/Pkey(\freq));
    // random drift of grain duration
Pbindef(\grain, \sustain, Pbrown(1, 100, 3) /Pkey(\freq), \dur, 0.1);
```

Very short grains seem softer than longer ones; one can try comparing two alternating grains and adjusting their amplitudes until they seem equal.

```
// example 22.3 - loudness loss for short grains
(
Pbindef(\grain).clear;
Pbindef(\grain,
    \instrument, \gabor1,
    \freq, 1000,
    \dur, 1,
    \sustain, Pseq([0.001, 0.2], inf),
    \amp, Pseq([0.1, 0.1], inf)
).play;
)
(    // short grain 2x louder
Pbindef(\grain,
    \sustain, Pseq([0.001, 0.2], inf),
    \amp, Pseq([0.2, 0.1], inf)
);
)
(    // short grain 4x louder
Pbindef(\grain,
    \sustain, Pseq([0.001, 0.2], inf),
    \amp, Pseq([0.4, 0.1], inf)
).play;
)

    // a grain with quasi-rectangular envelope, short grain 6x louder.
(
SynthDef(\pip, { |out, freq=440, sustain=0.02, amp=0.2, pan=0|
    OffsetOut.ar(out,
    Pan2.ar(SinOsc.ar(freq)
    * EnvGen.ar(Env.linen(0.0005, sustain - 0.001, 0.0005, amp), doneAction: 2),
pan)
    );
}).memStore;

Pbindef(\grain).clear;
Pbindef(\grain,
    \instrument, \pip,
    \freq, 1000,
    \sustain, Pseq([0.002, 0.2], inf),
    \amp, Pseq([0.6, 0.1], inf)
).play;
)
```

Short silences imposed on continuous sounds have different effects dependent on the sound: While on steady tones, short pauses seem like dark pulses, and only longer ones seem like silences, short interruptions on noisier signal may be inaudible.

```
// example 22.4 - perception of short silences
p = ProxySpace.push;
~silence.play;
(
~source = { SinOsc.ar * 0.1 };
~silence = { |silDur=0.01|
    EnvGen.ar(
    Env([0, 1, 1, 0, 0, 1, 1, 0], [0.01, 1, 0.001, silDur, 0.001, 1, 0.01]),
    doneAction: 2) ! 2
};
~listen = ~source * ~silence;
~listen.play;
)
~silence.spawn([\silDur, 0.001]); // like an added pulse
~silence.spawn([\silDur, 0.003]);
~silence.spawn([\silDur, 0.01]);
~silence.spawn([\silDur, 0.02]);
~silence.spawn([\silDur, 0.04]);  // a pause in the sound
~source = { WhiteNoise.ar * 0.1 };
```

When granular sounds are played close together in time, it becomes difficult to tell in which order they occurred. Ex. 5 plays a high and a low sound together first, then makes the lag between them adjustable. Lags above 0.05 seconds are well audible, shorter ones become more difficult.

```
// example 22.5 - order confusion with sounds in fast succession.
// as grains move closer and closer together, their order becomes ambiguous.
(
            // 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).store;
)


(
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, 0.05);
Pbindef(\hi, \lag, 0.02);
Pbindef(\hi, \lag, 0.01);
Pbindef(\hi, \lag, 0.005);

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

    // is it easier when the sounds are panned apart?
Pbindef(\hi, \pan, 0.5); Pbindef(\lo, \pan, -0.5);
Pbindef(\hi, \pan, 0);   Pbindef(\lo, \pan, 0);
```

In fast sequences of granular sounds, their order is hard to discern, as the grains fuse into one sound object. But when the order changes, the new composite does sound different.

```
// example 22.6 - multiple grains fuse into one composite.
// when their order changes, the sound is different.
(
Pbindef(\grain4,
    \instrument, \percSin, \sustain, 0.03, \amp, 0.2,
    \freq, Pshuf([1000, 600, 350, 250]),
    \dur, 0.005
);
)
Pbindef(\grain4).play;
    // you cannot tell the order, but the composite sound
    // is different in character.
Tdef(\grain, { loop { Pbindef(\grain4).play; 1.wait } }).play;
    // change to one fixed order
Pbindef(\grain4, \freq, Pseq([1000, 600, 350, 250].scramble));
    // different order every time
Pbindef(\grain4, \freq, Pshuf([1000, 600, 350, 250]));
```

2 Grains and Clouds

Any sound particle shorter than e.g. 100 milliseconds (this is not a hard limit, just an order of magnitude) can be considered a grain, and could be used as an element in a group of sound particles; such groups may be called trains, if they are rather regular sequences, or clouds, if rather varied. We will first look at the details of single, simple grains, and then at the properties that arise as groups of them form streams or clouds.

2.1 Anatomy of a Grain

A grain is a short sound event consisting of a waveform, which can be generated synthetically, taken from a fixed waveform, or selected from recorded material (and possibly filtered)—and an envelope; this is an amplitude shape imposed on the waveform, which can strongly influence the sound character of the grain. To begin with, we restrict the examples here to simple synthetic waveforms, and experiment with the effects of different envelopes, waveforms, and durations on single grains. Figure 22.7 creates an envelope and a waveform signal as arrays; in order to impose the envelope shape on the amplitude of the waveform, they are multiplied, and the three signals are plotted (for the waveforms, see figure 22.8). Env.sine is very close to a gaussian envelope, so this is good approximation of a sound quantum as postulated by Gabor.

```
// figure 22.7 - waveform, envelope, grain
```

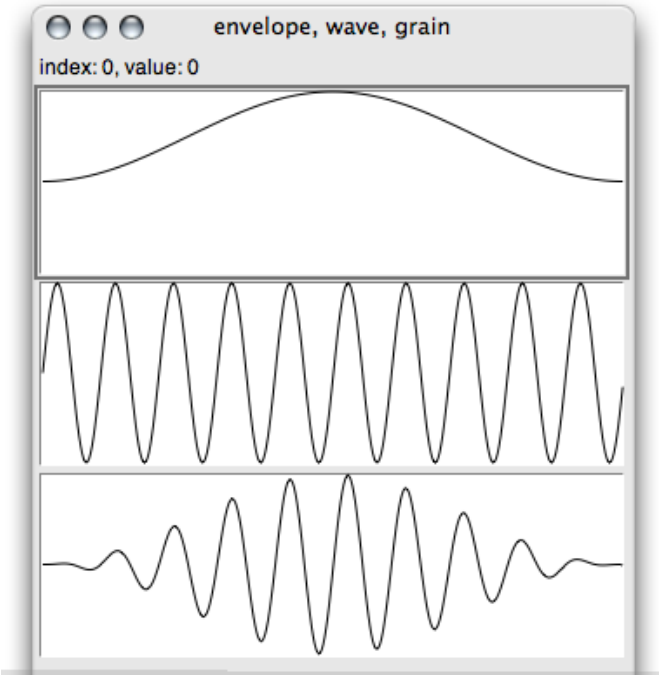
```
e = Env.sine.asSignal(400).as(Array);
w = Array.fill(400, { |i| (i * 2pi / 40).sin });
g = (e * w);

[e, w, g].flop.flat.plot("envelope, wave, grain", Rect(0,0, 408, 600), numChannels:
3);
```

Putting these elements together in a SynthDef allows for creating a variety of instances of one kind of grain by varying the frequency of the waveform, and grain duration, amplitude, and spatial position. The details in figure 22.9 show recommended practices for granular synthesis: Grains may have extremely short durations, so it is best to use audio rate envelopes. The timing between grains should be as accurate as possible, as the ear is very sensitive to micro-rhythmic variations (to be shown later), so one uses OffsetOut to allow for sample-accurate timing of the start of the grain's synthesis process. Also, one typically does not change grain synthesis parameters while the grain plays, so one can optimize efficiency by using \ir (initialisation rate) arguments. Finally, storing the SynthDef with .store or .memStore makes it available for use in patterns. The tests below play single grains both with Synth and as Events, using every parameter at least once to make sure they work correctly.

```
// figure 22.9
(
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)).memStore;
)

Synth(\gabor0);         // defaults from SynthDef
Synth(\gabor0, [\freq, 1000, \sustain, 0.005]);
Synth(\gabor0, [\freq, 1000, \sustain, 0.005, \amp, 0.1, \pan, 0.5]);

(instrument: \gabor0).play;                // default values from Event.
defaultEvent
(instrument: \gabor0, sustain: 0.02).play;
(instrument: \gabor0, sustain: 0.001, freq: 2500, amp: 0.05, pan: -0.5).play;
(instrument: \gabor0, sustain: 0.0003, freq: 2500).play;
(instrument: \gabor0, sustain: 0.0001, freq: 2500).play;
```

SC3 allows for very quick testing of envelope variants; figure 22.10 shows 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, letting much of the waveform character through; quasi-gaussian envelopes make grains louder by holding the waveform at full amplitude in the middle of its course, and Welch interpolation is similar; an exponential decay can create physical plausibility, as struck physical resonators decay exponentially; a percussive envelope with a controllable attack time can articulate the hardness of the initial attack, while a reverse exponential decay can be an intriguing special case of unnaturalness. More complex envelopes, such as the sinc function, can be created with sampled mathematical functions, and played with buffers.

```
// figure 22.10 -  more envelopes

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

     // 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(4);
a.plot(bounds: Rect(0,0,409,200), minval: -1, maxval: 1);

b = Buffer.sendCollection(s, a, 1);
```



By writing Synthdefs for some of these envelopes in figure 22.12, we can experiment with variations of the fundamental grain parameters: waveform frequency, envelope shape, and grain duration.

```
// figure 22.12
(    // a gabor (approx. gaussian-shaped) grain
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 * AmpComp.ir(freq) * 0.5),
doneAction: 2);
     OffsetOut.ar(out, Pan2.ar(snd * env, pan));
}, \ir ! 5).store;

          // 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 env = EnvGen.ar(Env([0, 1, 1, 0], [fadeT, holdT, fadeT], \sin),
     levelScale: amp * AmpComp.ir(freq) * 0.5,
     doneAction: 2);
     OffsetOut.ar(out, Pan2.ar(snd * env, pan));
}, \ir ! 5).store;

          // a simple percussive envelope
SynthDef(\percSin, { |out, amp=0.1, freq=440, sustain=0.01, pan|
     var snd = FSinOsc.ar(freq);
     var env = EnvGen.ar(
     Env.perc(0.1, 0.9, amp * AmpComp.ir(freq) * 0.5),
          timeScale: sustain, doneAction: 2
     );
     OffsetOut.ar(out, Pan2.ar(snd * env, pan));
}, \ir ! 5).store;

          // a reversed  percussive envelope
SynthDef(\percSinRev, { |out, amp=0.1, freq=440, sustain=0.01, pan|
     var snd = FSinOsc.ar(freq);
     var env = EnvGen.ar(
     Env.perc(0.9, 0.1, amp * AmpComp.ir(freq) * 0.5, 4),
          timeScale: sustain, doneAction: 2
     );
     OffsetOut.ar(out, Pan2.ar(snd * env, pan));
}, \ir ! 5).store;

          // an exponential decay envelope
SynthDef(\expodec, { |out, amp=0.1, freq=440, sustain=0.01, pan|
     var snd = FSinOsc.ar(freq);
     var env = XLine.ar(amp, amp * 0.001, sustain, doneAction: 2) * (AmpComp.
ir(freq) * 0.5);
     OffsetOut.ar(out, Pan2.ar(snd * env, pan));
}, \ir ! 5).store;

          // a reversed exponential decay envelope
SynthDef(\rexpodec, { |out, amp=0.1, freq=440, sustain=0.01, pan|
     var snd = FSinOsc.ar(freq);
     var env = XLine.ar(amp * 0.001, amp, sustain, doneAction: 2) * (AmpComp.
ir(freq) * 0.5);
     OffsetOut.ar(out, Pan2.ar(snd * env, pan));
}, \ir ! 5).store;
)
```

In order to access all parameters of the granular stream while it is playing, example 22.13 uses the Pbindef class (see also chapter

XXXX-JITP).

```
// example 22.13 - changing grain duration, frequency, envelope
(
Pbindef(\grain0,
     \freq, 400,
     \instrument, \gabor1,
     \sustain, Pn(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(0.003, 0.03) }));
Pbindef(\grain0, \sustain, Pfunc({ exprand(0.001, 0.1) }));

     // change grain waveform (sine) frequency
Pbindef(\grain0, \freq, 200);
Pbindef(\grain0, \freq, 500);
Pbindef(\grain0, \freq, 1200);
Pbindef(\grain0, \freq, 3000);
Pbindef(\grain0, \freq, Pn(Pgeom(100, 1.125, 40)));
Pbindef(\grain0, \freq, Pfunc({ exprand(400, 1200) }));
Pbindef(\grain0, \freq, Pfunc({ exprand(100, 10000) }));

     // 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));

Pbindef(\grain0, \amp, 0.7);
```

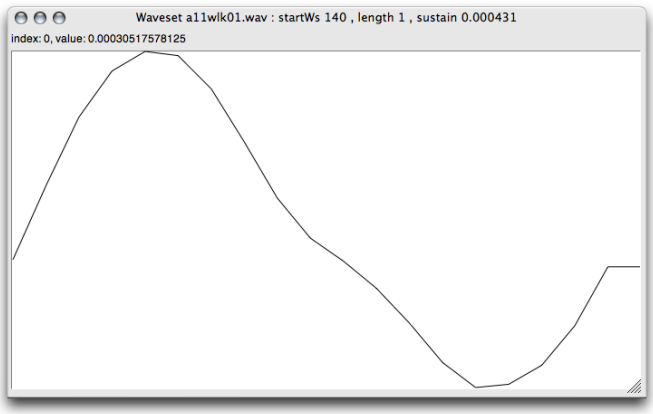
Parameter changes can have side effects of interest: For example, a rexpodec envelope ends with a very fast cutoff, and when the waveform at that point has a high amplitude, it creates a strong click transient. One can apply this for creating finer control on attack of a (forward) expodec grain: by making the phase of the oscillator adjustable, different attack colors can be articulated.

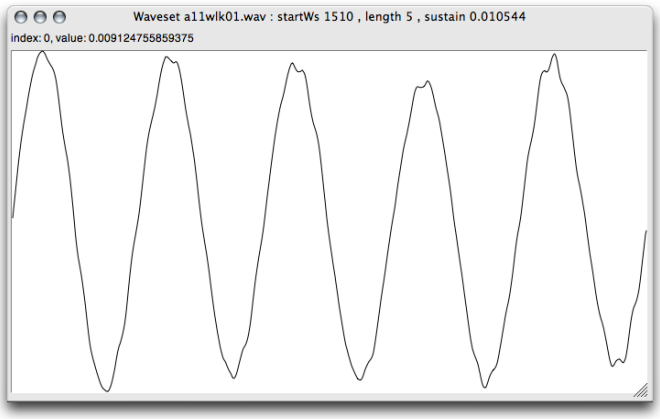
```
// figure 22.14 - adjusting phase for attack color

(    // an expodec envelope sine grain with adjustable phase
SynthDef(\expodecPH, { |out, amp=0.1, freq=440, click=0, sustain=0.01, pan|
     var snd = FSinOsc.ar(freq, click * 0.5pi);
     var env = XLine.ar(amp, amp * 0.001, sustain, doneAction: 2) * (AmpComp.
ir(freq) * 0.5);
     OffsetOut.ar(out, Pan2.ar(snd * env, pan));
}, \ir ! 6).store;
)
(
Pbindef(\grain0).play;
Pbindef(\grain0,
     \instrument, \expodecPH,
     \sustain, 0.1,
     \freq, [100, 300],
     \click, Pseq((0..20)/20, inf) // add more and more click
).play;
)
```

2.2 Textures, Masses, Clouds

When shifting attention from individual sound particles to the textures created by larger numbers of microsound events, the relations between aspects of the individual events in time create a number of emerging perceptual properties. Different terms have been created for these streams: very regular sequences of pulses may be called pulse trains; texture is a rather flexible general term used among others by Trevor Wishart (Audible Design, 1994); Edgard Varèse has often spoken of masses and volumina of sound (Varese 1962); the cloud metaphor suggests interesting vocabulary for imagining clouds of sound particles inspired by the rich morphologies of clouds both in the atmosphere and in interstellar space, and their evolution in time.

In synchronous granular synthesis, the particles occur at regular intervals and form 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 this case, density really becomes a statistical description of the average number of particles in a given time unit. Example 22.15 demonstrates four general strategies for controlling cloud parameters, here applied to cloud density:

- » fixed values—which creates a synchronous stream here;
- » time-varying values—specified by an envelope pattern here, creating accelerando and ritardando
- » random variation within ranges—with density, this creates a transition to asynchronous streams.
- » parameter dependent values—a cloud parameter is derived from another cloud parameter.

The generalization of this approach is called Grainlet synthesis (Roads 2001, pp. 125-129).

```
// example 22.15 - different control strategies for density

(    // 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, 0.04)
Pbindef(\grain0, \dur, 0.03)      // ...
Pbindef(\grain0, \dur, 0.025)
Pbindef(\grain0, \dur, 0.01)      // fundamental frequency 50 Hz

     // time-changing values: accelerando
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 * Pbrown(0.9, 1.1, 0.05))
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(0.6, 1.4))
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({ (0.2.linrand * 3) + 0.8 }));
Pbindef(\grain0, \dur, 0.02 * Pfunc({ (0.5.linrand * 3) + 0.5 }));
Pbindef(\grain0, \dur, 0.02 * Pfunc({ (1.0.linrand * 3) + 0.0 }));

    // very irregular timing - asynchronous
Pbindef(\grain0, \dur, 0.02 * Pfunc({ 2.45.linrand.squared }));


(   // duration dependent on freq parameter
Pbindef(\grain0,
    \freq, Pn(Penv([200, 1200], [10], \exp), inf),
    \dur, Pfunc({ |ev| 20 / ev.freq  })
);
)
(   // dependent on freq parameter, with some irregularity
Pbindef(\grain0,
    \freq, Pn(Penv([200, 1200], [10], \exp), inf),
    \dur, Pfunc({ |ev| 20 / ev.freq * rrand(0.5, 1.5)  })
);
)
```

The same strategies can be applied to any other cloud (or grain) parameters; in fact figure 22.15 has already employed a form of time-varying values, an envelope pattern. Example 22.16 shows how different control strategies can be applied to the set of grain and cloud parameters available so far.

```
// example 22.16 - control strategies applied to different parameters
(
Pbindef(\grain0).clear;
Pbindef(\grain0,
    \instrument, \expodec,
    \freq, 200,
    \sustain, 0.06,
    \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, 200 * Pwhite(-24.0, 24).midiratio);
    // timechanging with random variation
Pbindef(\grain0, \freq, Pn(Penv([200, 1200], [10], \exp), inf) * Pwhite(-24.0,
24).midiratio);

    // random spatial scattering
Pbindef(\grain0, \pan, Pwhite(-0.8, 0.8));
    // simple tendency
Pbindef(\grain0, \pan, Pn(Penv([-1, 1], [2]), inf));
    // coupled to other parameter - low freqs are left
Pbindef(\grain0, \pan, Pfunc({ |ev| ev.freq.explin(50, 5000, -1, 1) }));
    // reversed, and written with Pkey to access freq value
Pbindef(\grain0, \pan, Pkey(\freq).explin(50, 5000, 1, -1));

    // more time scattering variants
Pbindef(\grain0, \dur, 0.1 * Pwhite(0.5, 1.5));
Pbindef(\grain0, \dur, 0.05 * Prand([0, 1, 1, 2, 4], inf)); // rhythmic random

    // amplitude - randomized
Pbindef(\grain0, \amp, Pwhite(0.01, 0.2));
    // perceptually more even randomization - impression of 'depth'
Pbindef(\grain0, \amp, Pwhite(-50, -14).dbamp);
    // more depth by amplitude variation may sound sparser
    // and call for more density
Pbindef(\grain0, \dur, 0.025 * Prand([0, 1, 1, 2, 4], inf));

    // random tendencies for amplitude 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 depends on freq
Pbindef(\grain0, \sustain, Pkey(\freq).reciprocal * 20).play;
```

2.3 CloudGenMini

Finally, we come to a reimplementation of a classic microsound program written by Curtis Roads and John Alexander, CloudGenerator, which creates clouds of sound particles by means of statistical distributions in the form of random ranges and tendency masks. Here, the discussion focuses on the synthesis control techniques and the aesthetic aspects, for coding style aspects see the chapter XXXX-Object Modelling. (As the code is shown there, it is not repeated here.)

CloudGenMini combines several components: several synthdefs can be chosen that create different flavors of sound particles; a Tdef creates a cloud of grains, based on random values within ranges given for every parameter. New ranges can be created (also by random), stored, and crossfaded between. This allows for creating clouds based on tendency masks, which was a central feature of CloudGenerator. For playing CloudGenMini, a simple GUI is also provided.

The Tdef(\cloud0) function is 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 constructions 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 help with finding more interesting areas. CloudGenMini can create random ranges for all synthesis and cloud parameters, within global maximum settings, and can switch or interpolate between eight stored range settings.

CloudGenerator features tendency masks for creating clouds: Cloud duration, start and end values for high and low bandlimit (the minimum and maximum frequencies of the grain waveform), the grain duration, density, and amplitude can be specified to define a cloud's evolution in time. CloudGenMini generalizes this approach: By setting ranges for all parameters, and crossfading between these, every parameter can evolve from deterministic to random variation within a range; so, e.g. a densityRange of [10, 10] going to [1, 100] creates a synchronous cloud, which evolves to asynchronicity over its specified duration.

3 Granular Synthesis on the Server

So far, all the examples given have created every single sound particle as one synth process; SC3 also has a selection of UGens that implement granular synthesis on the sound server entirely. While one can obtain similar sounds either way, it is interesting to experiment how parameter control by UGens suggests different solutions, and leads to different ideas.

The first granular synthesis UGen in SC was TGrains, which does soundfile granulation (covered soon); since version 3.1 the UGens GrainSin, GrainFM, GrainBuf, GrainIn, and Warp1 are part of the SC3 distribution.

GrainSin creates a stream of grains with a sine waveform and a Hanning-shaped envelope, with grains being triggered by a control signal's transition from 0 or less to positive. Example 22.17 is a simple conversion of the GrainSin help file example to JIT style (see chapter XXXX-JITP for details). In short, a ProxySpace is an environment for NodeProxies, or placeholders for synthesis processes. NodeProxies can be changed and reconfigured very flexibly while running, which makes them ideally suitable for fluid exploration of synthesis variants.

```
// figure 22.17 - GrainSin.help example as nodeproxy
```

```
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 to control panning
     // use WhiteNoise and mouse y to control deviation from center
     freqdev = WhiteNoise.kr(MouseY.kr(0, 400));
     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 figure 22.18, an envelope is converted to a Signal, sent to a buffer, and the ~grain proxy is set to use that buffer number. A bufnum of -1 restores it to the default envelope buffer.

```
// figure 22.18 - switching envelopes while playing

q = q ? ();    // make a dict to keep things around
q.envs = ();   // e.g. some envelopes
q.bufs = ();   // and some buffers
               // make an envelope, and convert 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);

     // switch between built-in hanning envelope and custom - perc1
~grain.set(\envbuf, -1);
~grain.set(\envbuf, q.bufs.perc1.bufnum);
Apart from changing the parameter controls of the proxy to fixed values, one can
also map control proxies to them. Figure 22.19 shows creating control signals as
proxies, and rewriting them while playing.
// figure 22.19 - fixed parameters and control proxies

~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) };
~grdur = { SinOsc.kr(0.3).range(0.01, 0.1) };
~grdur = 0.01;

     // 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);
~grdensity = { 2 ** LFNoise0.kr(1).range(2, 4) };
```

The GrainFM ugen introduces a variant for the synthesis process: As the name implies, it features a pair of sine oscillators to create frequency modulated waveforms in each grain. Example grainUgens.2a is based on the GrainFM help file example, rewritten as a nodeproxy. The second version of ~grain uses MouseX to control modulation range instead of panning; such rewrites are often useful for learning how different controls affect the sound.

```
// figure 22.20 - GrainFM with mouse control
(
~grain.play;

~grain = { arg envbuf = -1, density = 10, graindur=0.1, modfreq=200;
     var pan = MouseX.kr(-1, 1);
     var trig = Impulse.kr(density);
     var freqdev = WhiteNoise.kr(MouseY.kr(0, 400));
     var freq = 440 + freqdev;
     var moddepth = LFNoise1.kr.range(1, 10);
     GrainFM.ar(2, trig, graindur, freq, modfreq, moddepth, pan, envbuf)
     * 0.2
};
)
(    // rewrite to use mouseX for modulation range
~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 freq = 440 + freqdev;
     var modrange = MouseX.kr(1, 10);
     var moddepth = LFNoise1.kr.range(1, modrange);
     GrainFM.ar(2, trig, graindur, freq, modfreq, moddepth, pan, envbuf)
     * 0.2
};
)
```

For more flexibility in experimentation, one can convert the controls of interest to proxies, and access them in the main proxy, here ~grain. This allows for changing controls individually, and even crossfading between old and new control synthesis functions. Example 22.21 is such a rewrite, where all parameters can be changed freely between fixed values and synthesis functions, line by line, in any order.

```
// figure 22.21 - GrainFM with individual control proxies
(
~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;
     var pan = WhiteNoise.kr;

     GrainFM.ar(2, ~trig.kr, ~graindur.kr,
     ~freq.kr, ~modfreq.kr, ~moddepth.kr,
     pan, envbuf) * 0.2
};
~grain.play;
)
     // change control ugens:
     // modfreq roughly follows freq
~modfreq = { ~freq.kr * LFNoise2.kr(1).range(0.5, 2.0) };

     // random triggering, same density
~trig = { |dens=10| Dust.kr(dens)};

~freq = { LFNoise0.kr(0.3).range(200, 800) };
~moddepth = 3; // fixed depth
~graindur = { LFNoise0.kr.range(0.01, 0.1) };

     // blend dust and impulse triggers
~trig = { |dens=20, bal=0.2| Dust.kr(dens * (1-bal)) + Impulse.kr(dens * bal) };
~trig.set(\bal, 0.2);
~trig.set(\bal, 0.5);
~trig.set(\bal, 0.8);
```

Finally, we look at the GrainBuf ugen, which takes its waveform from a buffer on the server. Typically, this is used for soundfile granulation. Here, one potentially gets variety and movement in the sound stream by constantly moving the file read position (i.e. where in the soundfile to take the next grain waveform from), and varying the playback rate. Even simply moving the file read position along the time axis can create interesting articulation of the granular stream. Example 22.22 is a rewrite of the GrainBuf help file example with separate control proxies, and demonstrates a self-moving read position, controlling the read position by mouse, and the influence of asynchronous triggering, grain duration, and envelope shape on the sound.

```
// figure 22.22 - GrainBuf and control proxies

q.bufs.apollo = Buffer.read(s, "sounds/a11wlk01-44_1.aiff");
(
~grain.set(\wavebuf, q.bufs.apollo.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 = 7;
     var pan = WhiteNoise.kr;
     GrainBuf.ar(2, ~trig.kr, ~graindur.kr,
     wavebuf, ~rate.kr, ~filepos.kr, 2,
     pan, envbuf) * 0.2
};
~grain.play;
)

~trig = { |dens=20| Impulse.kr(dens) };
~graindur = 0.1;
~rate = { LFNoise1.kr.range(0.99, 1.01) };
~filepos = { MouseX.kr + LFNoise0.kr(100, 0.03) };

~graindur = 0.06;
~trig = { |dens=50| Dust.kr(dens) };
~grain.set(\envbuf, q.bufs.perc1.bufnum); // made in figure 22.18
~grain.set(\envbuf, -1);
```

TGrains, the first granular ugen in SC, is very similar, except that it only has a fixed envelope shape; GrainIn is also very similar, but gets the grain waveform from any running input signal, and uses a buffer envelope; employing the built-in multichannel panning (PanAz-like) that this family of UGens share, it can be used elegantly for spatial scattering of (e.g. continually processed) live input.

When comparing these UGen-based and individual-grain particle synthesis, one can say that experimenting with UGens as

controls allows for very interesting behavior; even much pattern-like behavior can be realized with Demand UGens (see the Demand help file). On the other hand, one cannot write one's own synthesis variant for a special kind of grain in SC3 directly in server-side granular synthesis. Unless one wants to implement new UGens in C++ (see chapter XXXX-UGens), one is limited to the synthesis processes which exist as granular UGens.

4 Exploring Granular Synthesis Flavors

The most flexible point of entry to creating one's own flavors of microsound is considering that any waveform can be inside a grain. Reviewing the synthdef given in example 22.9, all one needs to change is the sound source itself. As a first example, we replicate GrainFM and GrainBuf as synthdefs. In example 22.23, synthdef \grainFM0 adds three parameters and changes the sound source to an FM pair, while synthdef \grainFM1 switches to a buffer envelope, followed by some example patterns.

```
// example 22.23 - FM grain as synthdef
(
SynthDef(\grainFM0, {|out, carfreq=440, modfreq=200, moddepth = 1,
     sustain=0.02, amp=0.2, pan|

     var env = EnvGen.ar(Env.sine(sustain, amp), doneAction: 2);
     var sound = SinOsc.ar(carfreq, SinOsc.ar(modfreq) * moddepth) * env;

     OffsetOut.ar(out, Pan2.ar(sound, pan))
}, \ir.dup(7)).memStore;
)
(instrument: \grainFM0, sustain: 0.1, amp: 0.2).play;

     // to use buffer envelopes: Osc1
(
q = q ? ();
q.envbuf = Buffer.sendCollection(s, Env.perc(0.1, 0.9).discretize, 1);

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)).memStore;
)

(instrument: \grainFM1, sustain: 0.1, envbuf: q.envbuf).play;

(
Pbindef(\fm,
     \instrument, \grainFM1,
     \carfreq, Pbrown(300, 1200, 300),
     \modfreq, 200,
     \modindex, Pbrown(1.0, 10.0, 2.5),
     \sustain, 0.1,
     \dur, 0.1,
     \envbuf, q.envbuf,
     \pan, Pwhite(-0.8, 0.8)
).play;
)
```

GrainBuf is also straightforward to rebuild with a PlayBuf, as figure 22.24 shows. Both of them can then be played with patterns or tasks on the client side.

```
// figure 22.24 - GrainBuf as synthdef
(
q = q ? ();
q.envbuf = Buffer.sendCollection(s, Env.perc(0.1, 0.9).discretize, 1);
q.apollo = Buffer.read(s,"sounds/a11wlk01.wav");

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)).store;
)

(instrument: \grainBuf1, sustain: 0.1, envbuf: q.envbuf, wavebuf: q.apollo).play;

(
Pdef(\buf1,
     Pbind(
     \instrument, \grainBuf1,
     \envbuf, q.envbuf, \wavebuf, q.apollo,
     \sustain, 0.1, \dur, 0.05,
     \pan, Pwhite(-0.8, 0.8),
     \filepos, Pn(Pseries(0, 0.01, 100))
     )
).play;
)

q.envbuf.sendCollection(Env.sine.discretize);
q.envbuf.sendCollection(Env.perc.discretize);
```

Glisson synthesis is based on Iannis Xenakis' use of glissandi (rather than fixed-pitch notes) as building blocks for some of his instrumental music. In its simplest form, 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.

```
// figure 22.25 - Glisson synthesis
(
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);
     OffsetOut.ar(out,
               Pan2.ar(
               SinOsc.ar(XLine.ar(freq, freq2, sustain)),
               pan,
               amp
               ) * env
     )
}, \ir!7).store;
)

(
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, q.envbuf.bufnum
     ]);
     (3 / (i + 10)).wait;
     });
}).play;
)
```

One possibility for organizing glissando structures is magnetization patterns (Roads 2001, pp 121-125). Glissons may have shallow or wide glissando ranges, be uni- or bidirectional, and diverge from or converge toward a center frequency, see the example on the DVD.

Pulsar Synthesis is named after pulsars, spinning neutron stars discovered first 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, which is a central aspect of Pulsar Synthesis. It 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, which are both scaled to the pulse's duration. Pulsar synthesis has been designed by Curtis Roads in combination with a special control model: a set of tables which can be edited by drawing is used for designing both waveforms (for pulsaret and envelope) and a group of control functions for synthesis parameters over a given time; this concept has been expanded in the PulsarGenerator program (written in SC2).

The main two control parameters are: fundamental frequency or (fundfreq), the rate at which pulses are emitted; and formant frequency (formfreq), a frequency that determines how fast the pulsaret and envelope are played back, which effectively is like a formant control. For example, at a fundfreq of 20 Hz, 20 pulses are emitted per second; at a formfreq of 100Hz, every pulse is scaled to 0.01seconds length, so within the 0.05 seconds for one pulsar period, the duty cycle where signal is present is only 0.01

seconds long. Each pulsar train also has controls for amplitude and spatial trajectory. Figure 22.26 shows creating a set of tables, and sending them to buffers.

```
// figure 22.26 - 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.plot(\pulsaret);
```

Figure 22.27 realizes one pulsar train with the GrainBuf ugen, playing fixed parameter values at first. Changing the parameters one at a time and crossfading between them demonstrates the effect of formfreq and fundfreq movements. Finally, replacing the controls with looping tables completes a minimal pulsar synthesis program.

```
// figure 22.27 - Pulsars as nodeproxies using GrainBuf
(
p = ProxySpace.push;

    // fund, form, amp, pan
~controls = [ 16, 100, 0.5, 0];

~test.set(\wavebuf, q.bufs.pulsaret.bufnum);
~test.set(\envbuf, q.bufs.env.bufnum);

~test = { |wavebuf, envbuf = -1|
    var ctls = ~controls.kr;
    var trig = Impulse.ar(ctls[0]);
    var grdur = ctls[1].reciprocal;
    var rate = ctls[1] * (1024 / 44100);

    var pulsars = GrainBuf.ar(2, trig, grdur, wavebuf, rate, 0, 4, ctls[3],
envbuf);

    pulsars;
};
~test.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 = (1024 / 44100) / looptime;
    A2K.kr(PlayBuf.ar(1, [\fund, \form, \amp, \pan].collect(q.bufs[_]),
    rate: rate, loop: 1));
};
)
```

Example pulsar3 shows sending different tables to the buffers. One could then proceed to making graphical drawing interfaces for the tables, and writing the changes in the tables to their buffers.

```
// figure 22.28 - make new tables and send them to buffers

    // make new pulsaret tables and send them to the buffer:
q.bufs.pulsaret.sendCollection(Array.linrand(1024, -1.0, 1.0));
q.bufs.pulsaret.read("sounds/a11wlk01.wav", 44100 * 1.5);
q.bufs.pulsaret.sendCollection(Pbrown(-1.0, 1.0, 0.2).asStream.nextN(1024));


    // make a new random fundfreq, 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);
```

PulsarGenerator realized aspects of advanced pulsar synthesis by a number of extensions:

Three parallel pulsar trains (sharing fundfreq, but with separate formfreq, amp and pan controls) were being driven from separate control tables. One could switch or crossfade between sets of tables. Both table sets and banks of table-sets could be saved to disk.

Pulsar masking was implemented in two varieties: a burst ratio specifies 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 allows for creating subharmonics to 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 pulses), 0.5 meaning play every pulse with a 50% chance, which can create interesting intermittency effects. Both can be implemented in a UGen graph by multiplying the trigger signal with demand UGens that provide a sequence, or a random decision to mute the trigger, e.g. with CoinGate.

Pulsar synthesis can also be implemented with client side control, as example pulsar4 shows: pulsaret and envelope are played with a synthdef, with a simple starting pattern.

```
// figure 22.29 - pulsar synthesis with client-side control

    // a pulsaret and an envelope
a = Signal.sineFill(1024, 1/(1..10).scramble);
b = Env.perc.discretize;

a.plot;
b.plot;
    // as buffers
x = Buffer.sendCollection(s, a, 1);
y = Buffer.sendCollection(s, b, 1);

    // a pulsar synthdef
(
SynthDef(\pulsar1, {|out, wavebuf, envbuf, form=200, amp=0.2, pan|
    var grDur =  1/form;
    var pulsaret = Osc1.ar(wavebuf, grDur);
    var env = Osc1.ar(envbuf, grDur, doneAction: 2);

    OffsetOut.ar(out, Pan2.ar(pulsaret * env, pan, amp));
}, \ir ! 6).store;
)
Synth(\pulsar1, [\wavebuf, x, \envbuf, y]);

    // a simple pattern
(
Pbindef(\pulsar1,
    \instrument, \pulsar1,
    \wavebuf, x, \envbuf, y,
    \form, Pn(Penv([20, 1200], [4], \exp)).loop,
    \amp, 0.2, \pan, 0,
    \fund, 12,
    \dur, Pfunc({ |ev| ev.fund.reciprocal })
).play;
)
```

One can then choose to control parameters from patterns such as envelope segment players, or also with table control, as seen in example pulsar5. This would allow for controlling 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.

```
// figure 22.30 - control from patterns and tables

Pbindef(\pulsar1, \form, Pn(Penv([20, 1200], [4], \exp)).loop);
Pbindef(\pulsar1, \fund, Pn(Penv([5, 50], [5])).loop);
Pbindef(\pulsar1, \amp, Pn(Penv([0, 0.2, 0.1], [7])).loop);
Pbindef(\pulsar1, \pan, Pbrown(-1.0, 1.0, 0.2));
Pbindef(\pulsar1, \amp, 0.2);

    // use Pseg for a control table reader
f = Env({ 100.0.rand }.dup(10), {1.0.rand}.dup(9).normalizeSum, \sine).asSignal;
f.plot;

Pbindef(\pulsar1, \fund, Pseg(f, 0.01, \lin, inf)).play;
```

Both, Curtis Roads and Florian Hecker realized a number of works with material generated by pulsar synthesis. Pulsars can be used particularly well as exciter signals for filters, or as input material for convolution processes (Advanced Pulsar Synthesis, Roads 2001 pp.147-154). Tommi Keränen has realized an SC3 version of PulsarGenerator with a slightly different feature set,

which has not been made officially available.

5 Soundfiles as Microsound Material

Soundfiles 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 (see e.g. figure 22.22, GrainBuf, earlier). However, both writing synthdefs for granulating soundfiles, and analysing the internal structure of soundfiles for use as a source of microstructure provide more areas to explore. We present examples for both: constant Q granulation, and Wavesets.

5.1 ConstantQ Granulation

ConstantQ granulation extends soundfile granulation by bandpass filtering each grain, and letting the filter ring long enough to let it decay. Figure 22.31 shows a synthdef for it: a grain is read around a center position within the file (as in TGrains, in seconds); ringtime and amplitude compensation for the given frequency and rq are estimated; after ringtime is over, synthesis is faded out by a cutoff envelope.

```
// figure 22.31 - SynthDef for constant Q granulation.

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

     var envSig = EnvGen.ar(Env([0, amp, 0], [0.5, 0.5] * sustain, \welch));
     var grain = PlayBuf.ar(1, bufnum, rate, 0,
     centerPos - (sustain * rate * 0.5) * BufSampleRate.ir(bufnum),
     1) * envSig;

     var ringtime = (2.4 / (freq * rq) * 0.66).min(0.5); // estimated
     var ampcomp = (rq ** -1) * (400 / freq ** 0.5);

     var cutoffEnv = EnvGen.kr(Env([1, 1, 0], [sustain+ringtime,0.01]),
doneAction: 2);


     OffsetOut.ar(out,
     Pan2.ar(
               BPF.ar( grain, freq, rq, ampcomp ),
               pan,
               cutoffEnv
     )
     )
}, \ir.dup(10)).memStore;

)
```

Figure 22.32 shows test for the parameters: which region of the file to access, exciter grain duration; high rq values only color the grain a little, low rq values create definite ringing pitches. Because q is constant, lower frequencies will lead to longer ringtimes.

```
// figure 22.32 - parameter tests for constant Q granulation

Synth(\constQ, [\bufnum, b]);
Synth(\constQ, [\bufnum, b, \centerPos, 0.5]);      // centerPos = where in
soundfile (seconds)
Synth(\constQ, [\bufnum, b, \centerPos, 1]);
Synth(\constQ, [\bufnum, b, \centerPos, 1.5]);

     // sustain is sustain of exciter grain:
Synth(\constQ, [\bufnum, b, \centerPos, 0.5, \sustain, 0.01]);
Synth(\constQ, [\bufnum, b, \centerPos, 0.5, \sustain, 0.03]);
Synth(\constQ, [\bufnum, b, \centerPos, 0.5, \sustain, 0.1]);
Synth(\constQ, [\bufnum, b, \centerPos, 0.5, \sustain, 0.3]);

     // filter parameters, rq of bandpass implies ringtime
Synth(\constQ, [\bufnum, b, \freq, 1200, \rq, 1]);
Synth(\constQ, [\bufnum, b, \freq, 1200, \rq, 0.3]);
Synth(\constQ, [\bufnum, b, \freq, 1200, \rq, 0.1]);
Synth(\constQ, [\bufnum, b, \freq, 1200, \rq, 0.03]);
Synth(\constQ, [\bufnum, b, \freq, 1200, \rq, 0.01]);
Synth(\constQ, [\bufnum, b, \freq, 1200, \rq, 0.003]);

     // lower freq rings longer
Synth(\constQ, [\bufnum, b, \freq, 600, \rq, 0.003]);
```

Finally, figure 22.33 demonstrates creating a stream of constant Q grains with a Pbindef pattern, which allows for changing all parameter patterns while playing.

```
// figure 22.33 - a stream of constant Q grains
(
Pbindef(\gr1Q,
     \instrument, \constQ,
     \bufnum, b.bufnum,
     \sustain, 0.01,
     \amp, 0.2,
     \centerPos, Pn(Penv([1, 2.0], [10], \lin)),
     \dur, Pn(Penv([0.01, 0.09, 0.03].scramble, [0.38, 0.62] * 10, \exp)),
     \rate, Pwhite(0.95, 1.05),
     \freq, Pbrown(64.0, 120, 8.0).midicps,
     \pan, Pwhite(-1, 1, inf),
     \rq, 0.03
).play;
)
     // changing parameters while playing
Pbindef(\gr1Q, \rq, 0.1);
Pbindef(\gr1Q, \rq, 0.01);
Pbindef(\gr1Q, \sustain, 0.03, \amp, 0.08);
Pbindef(\gr1Q, \freq, Pbrown(80, 120, 18.0).midicps);
```

5.2 Wavesets and Granular Synthesis

Trevor Wishart introduced the Waveset concept in his book Audible Design (1994), implemented it in the CDP framework as a set transformation tools, and employed it in several of his compositions (e.g. in Tongues of Fire, 1994). While Wishart considers wavesets mainly as units for transformations of recorded sound material, it is also a strategy of turning a soundfile (or buffer) into a large repository of waveform segments that can be used in many different ways.

A waveset is defined as the waveform segment from one zero-crossing of a signal to the third, so in a sinewave, it would correspond to the sinewave's period. In a more aperiodic signal (such as a soundfile), waveform segments of widely varying length and shape can be found. The Wavesets class analyzes a soundfile into wavesets, and keeps the zero-crossings (from negative to positive), lengths, amplitudes, and some other values of every waveset.

The following lists Wishart's names for waveset transforms, with example descriptions. Many of them leave the original soundfile duration intact.

Transposition: take e.g. every second waveset, slow waveset playback down by e.g. two.

- » Reversal: play every waveset (or group of wavesets) time reversed
- » Inversion: turn every half-waveset inside out
- » Omission: drop every m out of n wavesets for silence (or also randomly by percentage)
- » Shuffling: switch every two adjacent wavesets (or groups of two)
- » Distortion: multiply by a power factor (i.e. exponentiate while keeping peak constant)
- » Substitution: take any other waveform (e.g. sine, square, other waveset) instead of waveset
- » Harmonic distortion: add double-speed and triple-speed wavesets on top of every waveset, weighted and summed.
- » Waveset averaging: scale adjacent wavesets to avg length and average them.

» Waveset Enveloping: put an envelope over single, or several wavesets.
» Waveset transfer: combine waveset timing from one source with waveset forms from another.
» Interleaving: take alternating wavesets (or groups) from two sources.
» Waveset time-stretching: repeat every waveset (or group) n times -> 'pitch beads'.
» Time-stretch with interpolation between adjacent wavesets.
» Time-shrinking : keep every n-th waveset or group.

Many of these can be demonstrated simply with the Wavesets class, one synthdef, and one Pbindef pattern. Figure 22.31 makes a waveset from a soundfile, and shows its internals; the lists of zero-crossings, lengths, amplitudes, minima, maxima, and fractional zerocrossings and lengths.

```
//   figure 22.34 - a Wavesets object
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.

     // some waveset statistics
w.minSet;             // shorted waveset
w.maxSet;             // longest waveset
w.avgLength;   // average waveset length
w.minAmp;             // softest waveset
w.maxAmp;             // loudest waveset
w.avgAmp;             // average waveset amplitude

     // show distribution of lengths
w.lengths.plot;
w.amps.plot;
Example 2 shows accessing individual wavesets or groups of wavesets and plotting
them.
//   figure 22.35 - accessing waveset data

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

     // 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 22.37 creates a second waveset, loads both into buffers on the server, and creates a synthdef to play wavesets with: bufnum is the buffer that corresponds to the waveset, start and length are the start frame and length of the waveset to play, and sustain is the duration for which to loop over the buffer segment. As the envelope cuts off instantly, one should calculate the precise sustain time and pass it in, as shown in the last section.

```
//   figure 22.37 - wavesets and buffers

     // make a second waveset, and 2 buffers for them
v = Wavesets.from("sounds/a11wlk01-44_1.aiff");
a = Buffer.read(s, "sounds/a11wlk01-44_1.aiff", 0, -1);
b = Buffer.read(s, "sounds/a11wlk01.wav", 0, -1);

     // A Synthdef to play a waveset (or group) n times.
(
SynthDef("waveset", { arg out = 0, bufnum = 0, start = 0, length = 1000,
     playRate = 1, sustain = 1, amp=0.2, pan;

     var phasor = Phasor.ar(0, BufRateScale.ir(bufnum) * playRate, 0, length) +
start;
     OffsetOut.ar(out,
     Pan2.ar(
               BufRd.ar(1, bufnum, phasor, interpolation: 4)
               * EnvGen.ar(Env([amp, amp, 0], [sustain, 0]), doneAction: 2),
               pan
               )
      );
}, \ir.dup(8)).store;
)

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

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

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

To show Wishart's transforms very simply, we use a Pbindef, so we can replace any patterns or fixed values in it while running. By default, the pbindef reconstructs part of the soundfile waveset by waveset.

```
// figure 22.38 - a pattern to play wavesets
     // create a pattern that plays a stream of wavesets;
     // by default, this reconstructs a soundfile segment as is.
(
Pbindef(\ws1).clear;
Pbindef(\ws1,
     \instrument, \waveset,
     \startWs, Pn(Pseries(0, 1, 3000), 1),
     \numWs, 1,
     \playRate, 1,
     \bufnum, b.bufnum,
     \repeats, 1,
     \amp, 1,
     [\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;
)
```

Now we can introduce waveset transposition, waveset reversal, simple time-stretching, waveset omission, and waveset shuffling in line by line examples.

```
//   figure 22.39 - some of wishart's transforms
// 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;
     // drop randomly by coin
Pbindef(\ws1, \numWs, 1, \freq, Pfunc({ if (0.25.coin, 1, \) }) ).play;
Pbindef(\ws1, \numWs, 1, \freq, 1, \startWs, Pn(Pseries(0, 1, 1000)) ).play; //
restore

     // waveset shuffle (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;
```

Waveset harmonic distortion can be realized very simply by playing a chord for each waveset, at integer multiples of the rate, and using appropriate amplitudes. Waveset interleaving only entails taking alternating Wavesets objects as sources, and getting waveset data and buffer numbers from them.

Waveset substitution requires a little more effort, as one needs to scale the substitute waveform into the time of the original waveform. Substituting a sinewave lets the time structure of the wavesets emerge, especially when each waveset is repeated. Note that figure 22.41 also considers the amplitudes for each waveset: as the substitute signal is full volume, scaling it to the original waveset's volume keeps the dynamic contour

intact. Finally, a brighter substitute waveform can have quite a different effect.

```
//   figure 22.41 - waveset substitution

     // the waveform to substitute
c = Buffer.alloc(s, 512);
c.sendCollection(Signal.sineFill(512, [1]));

(
Pbindef(\ws1).clear;
Pbindef(\ws1,
     \instrument, \waveset,
     \startWs, Pn(Pseries(0, 1, 1000), 5),
     \numWs, 1, \playRate, 1,
     \bufnum, 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;

     // and a different waveform to try
c.sendCollection(Signal.sineFill(512, 1/(1..4).squared.scramble));
c.sendCollection(Signal.sineFill(512, [1]));


c.plot;
```

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 most challenging and sonically most interesting. While one can imagine a number of ways of interpolation between two wavesets, here, the two wavesets are synched together: the sound begins with waveset1 at original speed and waveset2 scaled to the same loop duration; as the amplitude crossfades from waveset1 to waveset2, so does the loop speed, so that the interpolation ends with only waveset2 at its original speed.

```
//   figure 22. 42 - waveset interpolation
(
SynthDef("wsInterp", { arg out = 0,
     buf1 = 0, start1 = 0, len1 = 1000,
     buf2 = 0, start2 = 0, len2 = 500,
     playRate = 1, sustain = 1,
     amp=0.2, pan;

     var lenRatio = (len1 / len2);
     var playRateLine = Line.ar(playRate, playRate * lenRatio, sustain);

     var phasor1 = Phasor.ar(0, BufRateScale.ir(buf1) * playRateLine, 0, len1);
     var phasor2 = phasor1 / lenRatio;
     var xfade = Line.ar(0, 1, sustain);

     var snd = (BufRd.ar(1, [buf1, buf2],
     [phasor1 + start1, phasor2 + start2],
     interpolation: 4)
     * [1 - xfade, xfade]).sum;

     OffsetOut.ar(out,
     Pan2.ar(
              snd * EnvGen.ar(Env([amp, amp, 0], [sustain, 0]), doneAction: 2),
              pan
              )
      );
}, \ir.dup(12)).store;
)
```

```
(
q = q ? ();
q.playInterp = { |q, start1, len1, start2, len2, numWs=200|
     var set1 = w.frameFor(start1, len1).postln;
     var set2 = w.frameFor(start2, len2).postln;
     var sustain = (set2[2] + set1[2] * 0.5 * numWs).postln;

     (instrument: \wsInterp, buf1: b.bufnum, buf2: b.bufnum, amp: 0.5,

     start1: set1[0], len1: set1[1], playRate: 1,
     start2: set2[0], len1: set2[1], sustain: sustain
     ).play;
};
)
     // some interpolations
q.playInterp(200, 1, 500, 1, 400);
q.playInterp(400, 8, 600, 3, 100);
q.playInterp(200, 1, 500, 5, 600);
```

This transformation was also implemented in the MarcoHack software (Pranger 1999). The strategy shown can be extending into a pattern that plays a seamless stream of parameterizable interpolations.

Of course, there is no particular need to stick to waveset usage that is still recognizable as transformations of soundfiles. One can just as well begin with a very simple task that plays a single waveset as a granular stream, and add flexibility to it as ideas develop. Figure 22.43 is such a new start: it has a fixed grain repeat rate, a fixed starting waveset, and these are replaced with different values and later streams for generating values.

```
//   figure 22.43 - wavesets played with Tdef

     // 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: \waveset, 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);
```

In example 22.44, the waittime is derived from the waveset's duration, and a gap factor is added. All of these parameters are called with .next, so they can be directly replaced with infinite streams.

```
// figure 22.44 - derive waittime from waveset duration
(
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: \waveset, 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;
```

A wide range of possibilities opens here: one can create special orders of the wavesets; e.g. based on their lengths, or amplitudes; one could filter all waveset indices by some criterium, e.g. only keep the very soft ones.

To show just one example for modifying parameters based on waveset information, see figure 22.45: 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 it to make long wavesets short and vice versa. The pitchContour variable determines how drastically this transformation is applied. The example also shows waveset omission.

```
// figure 22.45 - add pitch contour and dropout rate
(
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: \waveset, 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

    // 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);
```

6 Conclusions

While many artists have realized works involving microsound in one way or another, the possibilities of new material using microsound as a resource is nowhere near being exhausted. Exploring recombinations and juxtapositions of synthesis approaches, and of strategies for structuring assemblages of microsound events, one can certainly find personal, idiosyncratic ways to create sounds. SuperCollider allows for many different working strategies: Whether one prefers writing scripts to create composed structures, or designing realtime instruments that can be played with hardware controllers, for creating sound material by playing, or for live performances, there are plenty of possibilities for further exploration.