Implementing Real-Time Granular Synthesis

Ross Bencina
Draft of 31st August 2001.

Overview

This article describes a flexible architecture for Real-Time Granular Synthesis that accommodates a number of Granular Synthesis variants including Tapped Delay Line, Stored Sample and Synthetic Grain Granular Synthesis in both Pitch Synchronous and Asynchronous forms. Three efficient algorithms for generating grain envelopes are reviewed and two methods for generating stochastic grain onset times are discussed. Readers are advised to consult the literature for further information on the theory and applications of Granular Synthesis.[1, 2, 3,4,5]

Background

Granular Synthesis or *Granulation* is a flexible method for creating animated sonic textures. Sounds produced by granular synthesis have an organic quality sometimes reminiscent of sounds heard in nature: the sound of a babbling brook, or leaves rustling in a tree. Forms of granular processing involving sampled sound may be used to create time stretching, time freezing, time smearing, pitch shifting and pitch smearing effects. Perceptual continua for granular sounds include gritty / smooth and dense / sparse. The metaphor of *sonic clouds* has been used to describe sounds generated using Granular Synthesis. By varying synthesis parameters over time, gestures evocative of accumulation / dispersal, and condensation / evaporation may be created.

The output of a Granular Synthesizer or *Granulator* is a mixture of many individual *grains* of sound. The sonic quality of a granular texture is a result of the distribution of grains in time and of the parameters selected for the synthesis of each grain. Typically, grains are quite short in duration and are often distributed densely in time so that the resultant sound is perceived as a fused texture. Algorithmic means are usually employed to determine when grains occur in time and to select their synthesis parameters, which often include duration, amplitude, panning, pitch, and envelope shape. A *Stochastic* Granular Synthesizer uses random number generators to determine grain onset times and synthesis parameters. Usually, the probability functions of these random number generators constitute the parameters of the Granulator as a whole and are often varied over time.

Granular Synthesis differs from many other audio synthesis techniques in that it straddles the boundary between algorithmic event scheduling and polyphonic event synthesis. Conventional synthesis techniques are typically employed in the creation of individual musical ‘notes’, whereas Granular Synthesis may employ thousands or millions of relatively simple grains to articulate a single sonic gesture. One implementation of Granular Synthesis using MIDI synthesizers was described as “Rapid Event Deployment” highlighting the central role of event scheduling in Granular Synthesis.[6]

Motivation

A Granulator generates individual grains by applying an *amplitude envelope* to a *source* such as a synthetic or sampled waveform. One criterion that has been used to classify different types of Granular Synthesis is the method used to generate the source for each grain. According to this classification scheme, types of Granular Synthesis include:

Tapped Delay Line Granular Synthesis, which uses a delay line to store samples from a real-time input stream. Each grain reads sound from the delay line with a potentially different delay time and playback rate. This form of Granular Synthesis is appropriate for ‘effects’ processing of real-time input.

Stored Sample Granular Synthesis, which generates each grain by reading sample values from a wavetable. It can be used to process pre-generated or sampled sounds.

Synthetic Grain Granular Synthesis describes a broad class of Granular Synthesizers that employs standard synthesis techniques such as oscillator or FM synthesis to synthesize each grain, thus generating purely synthetic sound textures.

Both Tapped Delay Line and Stored Sample Granular Synthesizers require information such as a wavetable or delay line to be shared between all grains. They also use interpolated sample lookup whose complexity is dependent on the desired interpolation quality and the efficiency of the interpolation implementation. Maintenance of the delay line and implementation of variable rate delay taps for each grain results in Delay Line Granulation having greater implementation complexity than the other types of Granular Synthesis described here. Synthetic Grain Granular Synthesis has the potential to be simpler than Delay Line and Stored Sample Granular Synthesis as no state is shared between grains. However, its implementation complexity is primarily dependent on the synthesis method used as the source of each grain.

Various algorithms for grain amplitude envelope generation and grain onset sequencing may be employed interchangeably in the implementation of the aforementioned types of Granular Synthesis. The following envelope algorithms can each be implemented with similar efficiency: Parabolic, Trapazoidal, and Raised Cosine Bell. Another common but possibly less efficient method of generating grain amplitude envelopes is the use of a lookup table containing the envelope function. Selection of a particular envelope algorithm is often based on the level of flexibility desired in specifying the envelope's attack, sustain and decay characteristics. In some applications such as analysis/resynthesis the spectral characteristics of the grain envelope may determine which algorithm is chosen.

Granular Synthesizers can also be classified in terms of the method used to distribute grains in time:

Asynchronous Granular Synthesis distributes grains in time according to a stochastic function that is traditionally specified in terms of *grain density:* the number of grains per unit time.

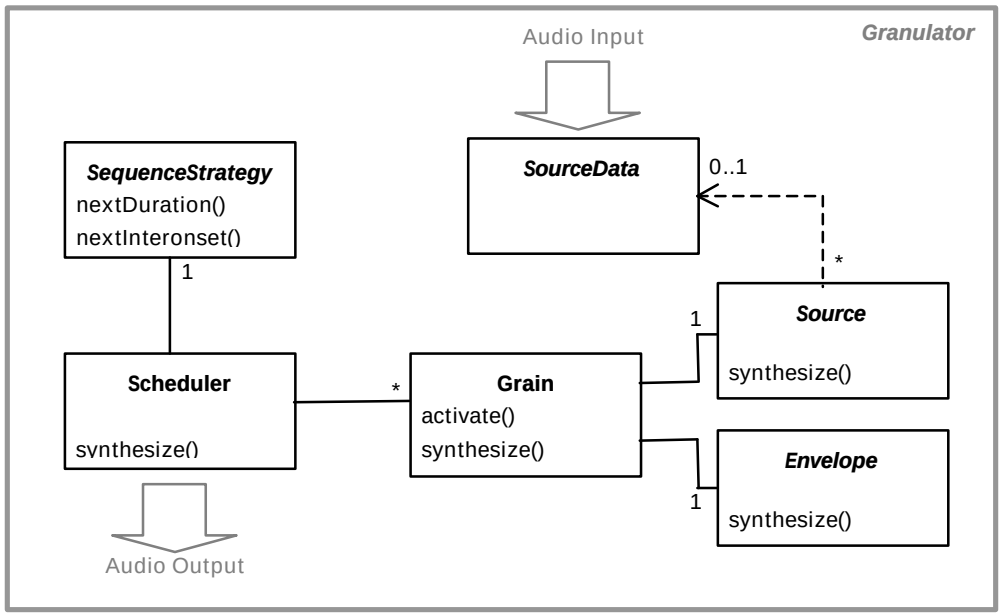
Pitch Synchronous Granular Synthesis sequences grains at regular intervals in time to create periodic or pitched sounds.

For real-time operation, where it is necessary to schedule grains in time-sequential order, a useful concept is *interonset time*: the time between the onset of successive grains. When implementing Asynchronous Granular Synthesis it is possible to generate interonset times as a function of grain density. In Pitch Synchronous Granular Synthesis the interonset time is directly related to the period of the resultant sound.

A Granulator may be required to synthesize thousands of grains per second, with common grain durations falling in the range from 10 to 70 milliseconds. Although individual grains are often simpler to synthesize than traditional synthetic "notes," the large number of grains involved in a typical granular texture often leads to significant processing overhead in scheduling, generating and passing parameters for each grain. Implementations of Granular Synthesis which utilise the standard event sequencing and scheduling facilities of general-purpose synthesis systems can exhibit poor or non real-time performance due to the overhead involved in initialising and passing parameters to each grain. Unifying grain scheduling, parameter generation, and synthesis within a Granular Synthesizer can significantly reduce this overhead, resulting in significant efficiency improvements over implementations where these functions are performed at different structural levels of a program, or by separate hardware systems.

An efficient implementation of Granular Synthesis requires grain scheduling, synthesis and mixing to be executed with a minimum of overhead. It is desirable to employ an architecture that accommodates the various grain source synthesis types, envelope algorithms and grain scheduling policies without forfeiting efficiency.

Structure



The diagram above shows the main participants and associations in a Granulator and indicates that SequenceStrategy, SourceData, Source and Envelope are abstract.

Participants

- **Granulator** (DelayLineGranulator, StoredSampleGranulator, SyntheticGrainGranulator)
  - Top level container responsible for the lifetime of its component parts.
  - Mediates between external audio streaming services, the Scheduler and possibly the SourceData, for example a Delay Line in the case of a Delay Line Granulator.
  - Manages dynamic parameter modulation or acts as a Facade[7] to allow clients to modulate synthesis parameters.

- **SequenceStrategy**
  - Provides an interface allowing the Scheduler to determine when the next grain should occur and what its duration should be.

- **Scheduler**
  - Maintains state necessary for activating grains according to Grain onset times and durations supplied by a SequenceStrategy.
  - Exposes a method for synthesizing samples of sound by mixing together the output from its active Grains.
  - Manages grain allocation, for efficiency it may maintain a pool of reusable Grains.

- **Grain**
  - Provides an interface for activation, synthesizing samples, and querying whether the grain has completed.
  - Contains an Envelope and Source, which are used by the Grain's synthesis method.

- **Envelope** (TrapezoidalEnvelope, ParabolicEnvelope, RaisedCosineBellEnvelope)

  - Synthesizes an amplitude envelope.

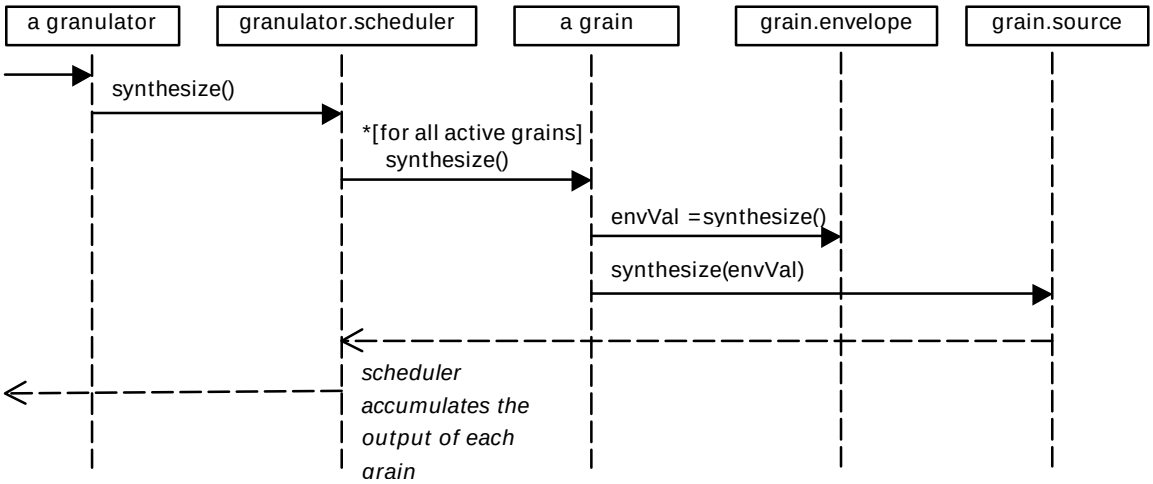
- **Source** (DelayLineSource, SampleSource, SyntheticSource)
  - Synthesizes the source waveform for a grain.
  - Maintains a reference to the Granulator's SourceData in Granulator variants where SourceData is required.

- **SourceData** (DelayLine, StoredSample)
  - Maintains shared state used by all instances of Source within the Granulator: a delay line in the case of a Delay Line Granulator, a wavetable in the case of a Stored Sample Granulator.

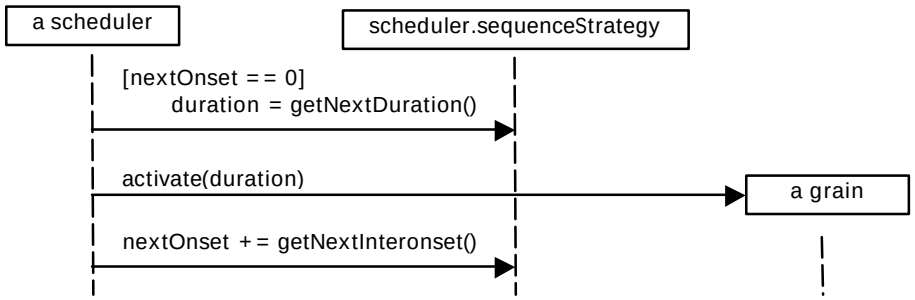
Collaborations

- In response to a request from an audio streaming service, Granulator requests that Scheduler synthesizes samples of sound.
- Scheduler synthesizes sound by requesting each of its active Grains to synthesize sound. The result of each Grain's output is mixed and returned to the client.
- Grain synthesizes samples in response to requests from the Scheduler. This is achieved by requesting that Envelope generate an amplitude value, which is then used by Source to synthesize a sample at the appropriate amplitude.
- Scheduler requests a grain interonset time and grain duration from SequenceStrategy and uses these to determine when the next grain should be activated (the next grain onset time) and its duration.
- At the next grain onset time, Scheduler activates a new Grain.

The following diagram shows the interaction that occurs when samples are requested from a Granulator. It illustrates how Grain obtains the current envelope value from Envelope and passes it to Source, while avoiding any implementation dependence between Source and Envelope.



The diagram below shows the interaction that occurs between Scheduler, SequenceStrategy, and Grain, when the Scheduler determines that it is time to activate a new Grain. Notice that SequenceStrategy is responsible for determining the duration of the activated Grain, and for determining when the next grain will occur.



Consequences

The motivation section identified a number of properties that differentiate various types of Granular Synthesis. These included the grain source synthesis method, the grain envelope generation algorithm, and the algorithm used to determine grain onset times. The structure described above allows these algorithms to be varied independently by encapsulating the sources of variation in the three abstract classes: Source, Envelope and SequenceStrategy.

More subtle benefits of this design arise from the allocation of responsibilities between participants; Grain passes the current envelope value to Source to control Source's output amplitude, this also allows Source to utilize the envelope value for additional purposes. For example, the value can be used as an FM modulation index in Synthetic Grain Granular Synthesis.[8] SequenceStrategy controls both Grain duration and onset time enabling grain sequencing algorithms where grains must precisely overlap, such as time domain pitch shifters and Pitch Synchronous Granular Synthesis.

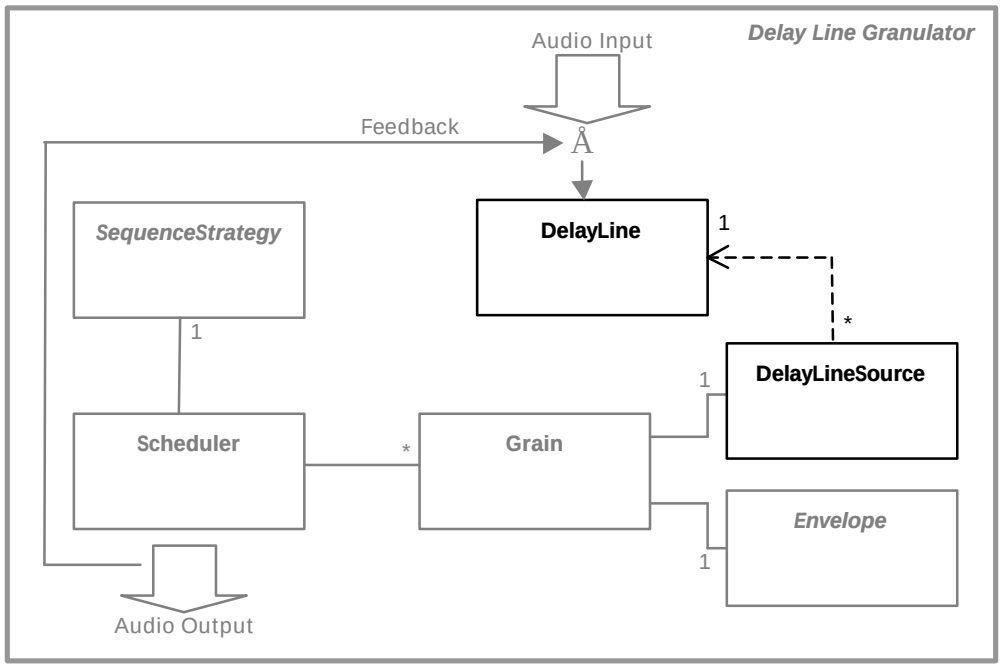
This design leaves open the problem of selecting individual synthesis parameters for each activated grain. Depending on the degree and flexibility of parameterization required, Source and Envelope, the Scheduler, or an additional object could be assigned this responsibility. A subsequent section describes a flexible approach that allows the grain parameter generation strategy to be varied independently of Envelope and Source.

The primary strength of this design is also its greatest weakness: it uses interchangeable, largely autonomous building blocks with little centralized control. This decomposition is well suited to common forms of Granular Synthesis such as Stochastic Granular Synthesis, where the random generation of each grain parameter is independent of all other grain parameters. However, it may be less appropriate when correlated grain parameters are required. For example, SequenceStrategy determines grain onset times and durations but has no control over grain pitch or transposition, thus making it difficult to implement a Granulator that can play rhythmic melodies.

Granulator Variants

The following subsections describe three different types of Granular Synthesis using the architecture presented above. Note that these Granulators vary only in terms of Source, SourceData, and external signal routing topology. They can be implemented using any of the Envelope and SequenceStrategy algorithms described later. Additional notes germane to the implementation of each type of Granulator are provided where relevant.

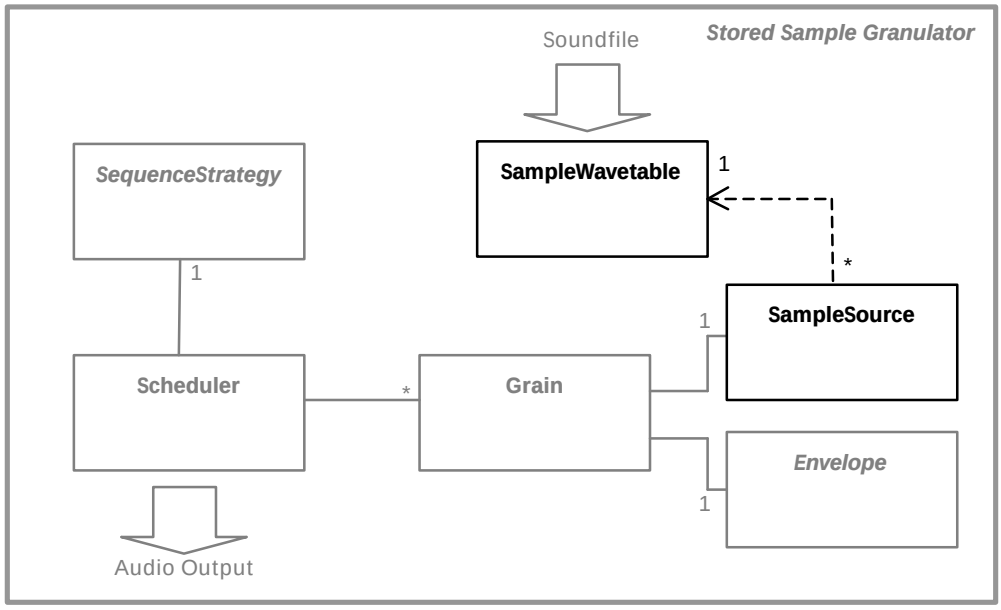
Delay Line Granulator



Each Grain Source in a Delay Line Granulator implements an independent delay tap on a Delay Line shared by all Grain Sources. Each delay tap may read the delay line from a different position to create delay and/or time smearing effects, and at a different rate to create pitch shifting and/or pitch smearing effects. Writing to the delay line may be temporarily halted to facilitate ‘time freezing’. If a delay tap has a playback rate greater than unity, care must be taken to avoid the non-causal case of trying to read ‘future samples’ from the delay line. One solution to this problem is to increase the initial delay time according to the requested grain duration and playback rate.

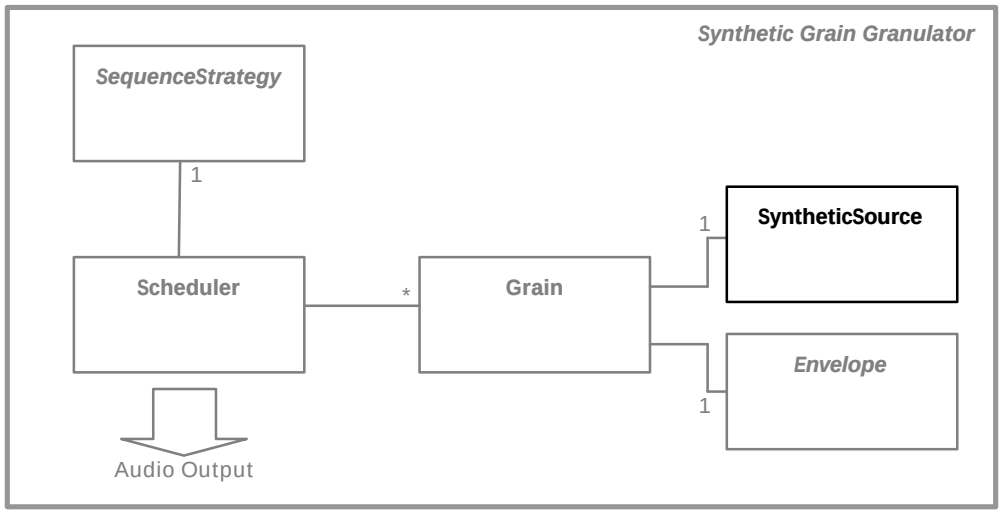
The output of the Delay Line Granulator may be mixed back into the delay line input to create feedback effects, which vary in sonic quality according to the granulation parameters. For example, feedback combined with pitch shifted grains creates stacked transpositions (chords) spaced according to the transposition factor. Due to the non-linear time and amplitude response of the sum of active grains it may be necessary to insert a compression or limiting element in the feedback loop to avoid instability.

Stored Sample Granulator



Grain Sources in a Stored Sample Granulator read sound from the stored sample at independent positions and rates. Ramping the initial read position slowly across the stored sample for successive grains creates time stretching or time dilation effects. Each Source's initial read position can be modulated by a small random factor to decorrelate source phases and create a more animated timbre.

Synthetic Grain Granulator



Grain Sources in a Synthetic Grain Granulator are responsible for synthesizing their waveform without any reference to a SourceData object. The available choices for SyntheticSource synthesis method are as limitless as the range of known sound synthesis methods. Traditionally, performance concerns have lead to the use of simple techniques such as single modulator FM or two oscillator additive synthesis.

Enhancements

Enhancements to the above types of granular synthesis include: Independent filtering (such as bandpass filtering) of each grain; Independent panning of each grain to create spatially diffused textures; Ramping the source playback rate of each grain to create glissandi grains or 'chirps.'[9]

Grain Envelopes

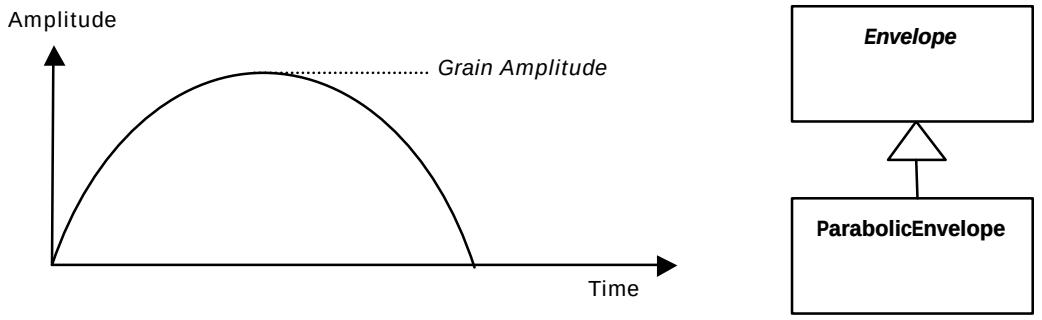
A number of factors can influence the choice of grain envelope algorithm including spectral response, flexibility of available envelope shapes and efficiency of generation.

Due to the short duration of individual grains, amplitude envelopes may introduce noticeable spectral artifacts, especially when grain durations are less than 10 to 15 milliseconds. Of the envelope algorithms presented below, trapezoidal envelopes introduce the greatest spectral distortion due to the second order discontinuities at each envelope state transition - raised cosine bell envelopes provide decreased spectral distortion at the cost of increased processing time. A formal analysis of the spectral artifacts introduced by various grain envelopes will not be presented here, however it is interesting to note that a related synthesis technique *FOF synthesis* uses the spectral effects of a specific grain envelope shape to simulate vocal tract formants.[10]

Although classical Granular Synthesis utilizes symmetrical grain envelopes, there are advantages to providing flexible grain envelopes with parameters to control envelope shape. For example, these parameters can be randomized to decorrelate artifacts introduced by the envelope. When the grain density is low and grain durations are relatively long, variations in envelope shape are easily perceived. As a result, a flexible grain envelope may be musically useful; for example a granular texture consisting of grains with slow attacks and fast decays can be made to sound like reversed tape playback.

The three envelope generation algorithms presented below may be implemented using iterative algorithms that employ a small number of state variables. This enables efficient implementation on modern processor architectures where memory access overhead may discourage the use of stored tables for grain envelopes.

Parabolic Envelope



Parabolae provide smooth grain envelopes, however they only allow parameteric control over grain amplitude and duration. Successive samples of a parabolic envelope may be computed efficiently using the following pseudo-code:[11]

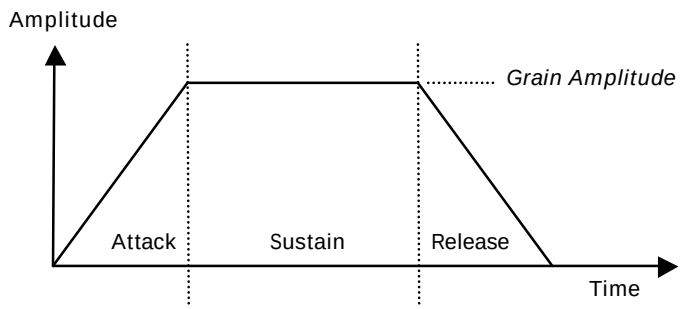
```
amplitude = amplitude + slope
slope = slope + curve
```

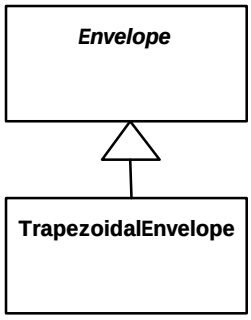
Initial values for `amplitude`, `slope` and `curve` may be computed as follows:

```
amplitude = 0
rdur = 1.0 / durationSamples
rdur2 = rdur * rdur
slope = 4.0 * grainAmplitude * ( rdur - rdur2 )
```

```
curve = -8.0 * grainAmplitude * rdur2
```

Trapezoidal Envelope





The trapezoidal envelope is considered a degenerate envelope due to the spectral artifacts introduced by its discontinuities. However it may be implemented efficiently, making it a candidate for real-time implementations. It provides the flexibility of independently variable attack, sustain and release times.

Successive samples of a trapezoidal envelope may be computed using an accumulator:

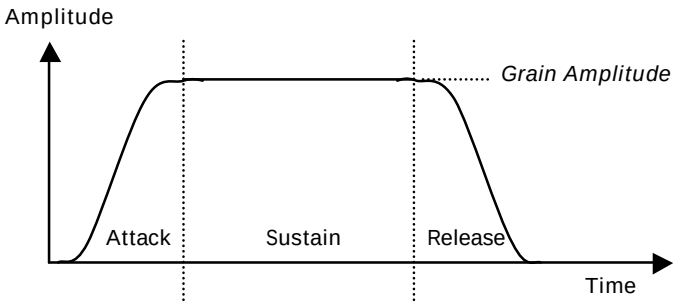
```
nextAmplitude = previousAmplitude + amplitudeIncrement
```

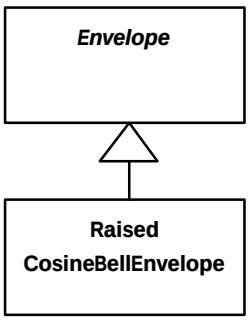
Where `amplitudeIncrement` varies according to the current envelope segment:

```
attack:
       amplitudeIncrement = grainAmplitude / attackSamples

sustain:
       amplitudeIncrement = 0

release:
       amplitudeIncrement = - ( grainAmplitude / releaseSamples )
```

`AmplitudeIncrement` need only be updated at envelope segment boundaries. A separate counter may be employed to determine when the next envelope segment boundary has been reached.

Raised Cosine Bell Envelope





The raised cosine bell envelope is a second-order continuous envelope which allows independent specification of attack, sustain and release times. The attack and release portions of the envelope are transposed cosinusoidal segments:

```
attack:
       amplitude = (1.0 + cos( PI + ( PI * ( i / attackSamples ) *
                                                (grainAmplitude / 2.0)
```

```
        sustain:
                amplitude = grainAmplitude

        release:
                amplitude = (1.0 + cos( PI * ( i / releaseSamples ) ) *
                                                        (grainAmplitude / 2.0)
```

Where `i` is the number of elapsed samples for the current envelope segment.

The following pseudo-code generates sinusoidal waveforms using only a single multiply-accumulate per sample; `y0` is the current value:[12]

```
        y0 = b1 * y1 – y2;
        y2 = y1;
        y1 = y0;
```

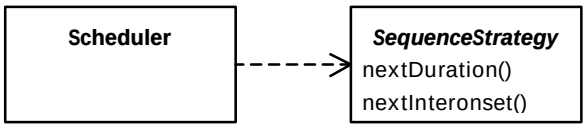
The state variables may be initialized as follows, where `w` is the phase increment and `ip` is the initial phase, both expressed in radians:

```
        b1 = 2.0 * cos( w )
        y1 = sin( ip - w )
        y2 = sin( ip – 2.0 * w )
```

This algorithm can be used to compute the cosinusoidal function in the attack and release portions of the Raised Cosine Bell Envelope. As with the Trapezoidal Envelope, state variables only need to be re-initialized at envelope segment boundaries.

Grain Scheduling

Traditionally grain onset times have been derived from a density parameter expressed as *grains per unit time*. Such temporally uniform grain distributions have a "natural" quality like rain on a tin roof. Non-real-time implementations can directly implement this type of distribution by allocating a buffer containing all output samples and randomly distributing grains across the buffer according to the required density.



For real-time operation new grains must be activated in time-sequential order. The basic variable controlling grain density in a real-time granulator is the *interonset time* – the time between temporally adjacent grain onsets. Using interonset times, a simple grain activation mechanism may be used whereby a counter is incremented by the next interonset time at each grain activation. For each subsequently generated sample the counter is decremented until it reaches zero at which point the next grain is activated and the process repeats:

```
float Scheduler::synthesize(){ // returns the output sample
        if( --nextOnset == 0 ){
                activateGrain( sequenceStrategy.nextDuration() );
                nextOnset += sequenceStrategy.nextInteronset();
        }

        return synthesizeActiveGrains();
}
```

In a real-time performance context it is sometimes desirable to preempt the `nextOnset` counter when the SequenceStrategy's interonset parameter(s) are reduced below the value previously issued by `nextInteronset()`.

Direct Interonset Specification

The simplest method of calculating grain interonset times in a real-time Stochastic Granulator is to express the interonset time as a random range:

```
interonset = minInteronset + ( frandom() * (maxInteronset – minInteronset) )
```

Where `frandom()` is a function returning real numbers from 0 to 1. As this method constrains interonset times to a specified range it is capable of creating subjectively 'smoother' fused textures than the density-per-unit time method described next. When the minimum and maximum interonset times are equal, grains are scheduled periodically, creating interesting amplitude-modulation style spectral effects.

Interonset Time as a Function of Density

To create classical granular textures with a specified grain density per unit time it is necessary to use a SequenceStrategy that generates interonset times as a function of density. The following pseudo-code generates interonset times based on an average grain density of D grains per second:[13]

```
interonsetTime = -log( frandom() ) / D
```

Where `log()` is the natural logarithm and `frandom()` is a function returning real numbers from 0 to 1.

Onset Quantization

An interesting extension to stochastic methods of grain onset scheduling is to quantize grains to(wards) fixed quantization intervals, thus creating rhythmic or 'pulsed' textures. This technique has been implemented in the author's AudioMulch software synthesizer.[15]

Limiting Grain Polyphony

In real-time contexts it may be necessary to constrain the number of processor cycles utilized by a granular synthesizer. An effective method of limiting processor usage is to place an upper bound on the number of simultaneously active grains – the *maximum grain polyphony.* The maximum grain polyphony can be hard-coded into the Scheduler, specified by the user, or dynamically varied according to processor usage information provided by the operating system.

Implementing a grain polyphony limit involves the Scheduler discarding any potential new grains, or deferring activation of new grains until existing grains have completed while the grain polyphony limit is exceeded. Traditional note based synthesizers sometimes limit polyphony using 'voice stealing,' whereby an active note (or grain in our case) is prematurely terminated to make way for a new note. However voice stealing is inappropriate for granular synthesis since a prematurely terminated grain is likely to create undesirable audible artifacts, whereas the perceptual effect of omitting a grain from a grain stream will often be negligible.

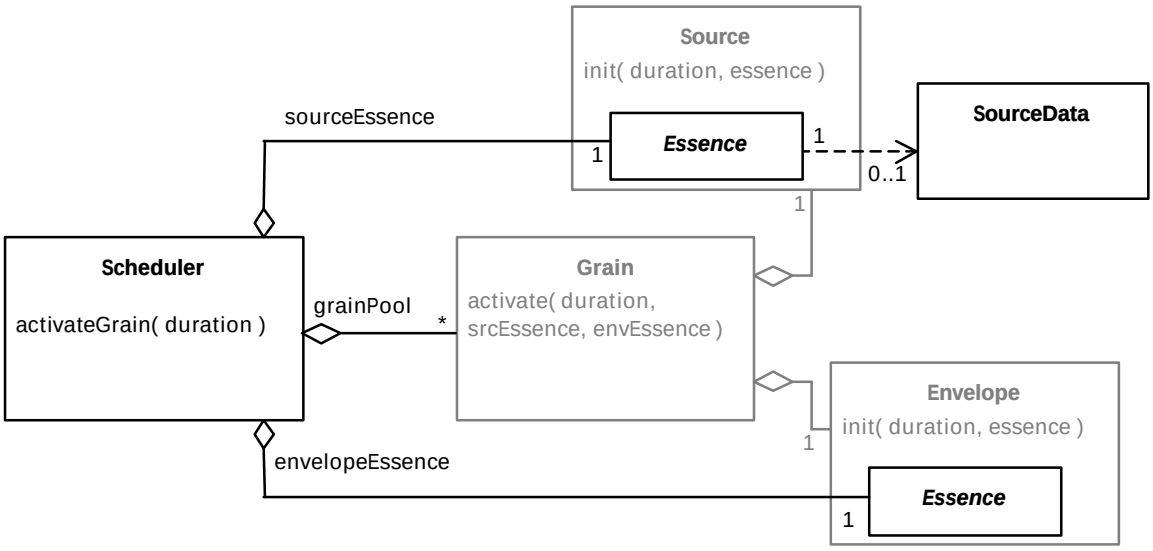
Initialising Grain State

The Granular Synthesizer architecture described earlier did not specify which object, or objects are responsible for generating synthesis parameters for newly activated Grains. This section presents a solution motivated by the following two design considerations:

Modularity: A variety of Source and Envelope algorithms are available, each of these algorithms requires a different set of initialization parameters. In the interest of allowing different algorithms to be used interchangeably, it is best to insulate other elements of the system such as the Scheduler and SequenceStrategy from the varying initialization requirements of each Source and Envelope algorithm.

Accessible Parameter Control: Initialization parameters for individual Grains are usually derived from parameters specified for the Granulator as a whole. For example, in Stochastic Granular Synthesis the parameters of individual grains are chosen randomly according to probability distribution parameters defined for the Granulator as a whole. These parameters are usually modulated by an external entity such a separate sequencing object or a graphical user interface. Thus it is desirable for all of the Granulator's parameters to be accessible from a singe location.

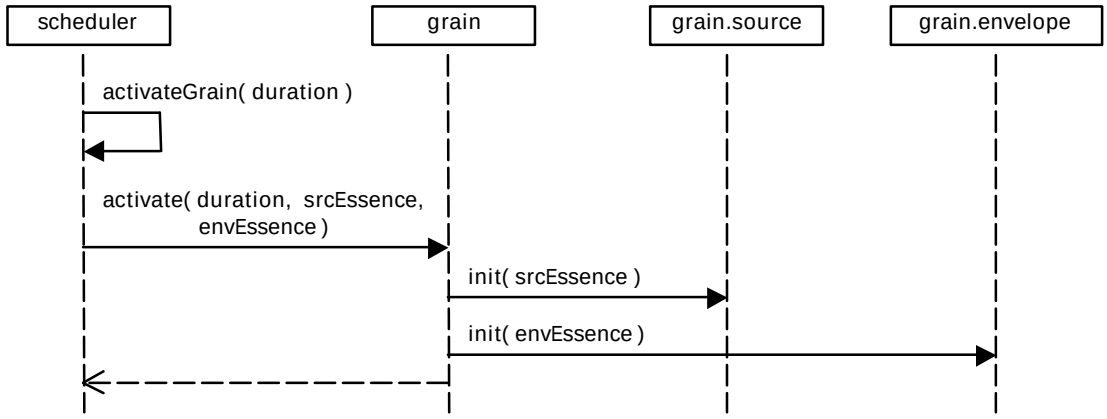
The method of initializing Grain state described below requires that each implementation of Source and Envelope define their own specialized initialization parameter type. For the remainder of this article these initialization parameter types are referred to as "Essences," after the design pattern of the same name. The Essence design pattern uses a separate object (the Essence object) to store initialization parameters for the object being created, in our case a Grain Source and Envelope.[15]



The diagram above illustrates the relationship between Scheduler, Grain, Source, Envelope and their respective Essences. Each concrete Source and Envelope class specifies a nested Essence type that reflects their specific initialization requirements. If Source and Envelope Essence are defined as abstract types multiple parameter generation strategies can be used with each concrete Source and Essence. If a separate abstract Essence hierarchy is constructed it may also be possible to use the same concrete Essence class with multiple Sources or Envelopes.

The Scheduler contains a single instance of Envelope::Essence and Source::Essence. Depending on the implementation, the Granulator, or an external client may access these Essences in order to modulate synthesis parameters. In the case of Delay Line and Stored Sample Granulators the Granulator may need to register the delay line or stored sample with scheduler.sourceEssence before synthesis begins. Alternately the source essence could be designed to directly contain the Delay Line or Stored Sample.

When the Scheduler needs to activate a new Grain it calls `Grain::activate()`, passing the grain duration along with references to envelopeEssence and sourceEssence. In turn, Grain passes the duration and respective essences to the `init()` methods of Source and Envelope enabling them to initialize themselves from parameters stored in their Essence. The sequence diagram below illustrates this process.



From a high-level perspective, grain activation is best viewed as object construction. However the high rate of grain allocation in a typical Granulator makes it desirable to use some form of object pooling to optimize grain activation and deactivation. The implementation described above uses the `Grain::activate()`, `Source::init()` and `Envelope::init()` methods to activate grains taken from a fixed pool. In C++, grain activation as object construction could be implemented by replacing `Grain::activate()`, `Source::init()` and `Envelope::init()` with constructor calls, and grain pooling could be implemented using the placement new operator.

Compile-Time Composition

Although the preceding discussion defined SequenceStrategy, Source, Envelope and their respective Essences as abstract or polymorpic types, they could also be considered as generic programming "concepts."[16] In the present context the distinction is primarily one of implementation methodology – by viewing them as concepts rather than abstract types a granulator may be efficiently implemented in C++ by using templates to create specialized versions of Grain and Scheduler at compile time:

```
template< class SourceT, class SourceEssenceT,
          class GrainEnvelopeT, class GrainEnvelopeEssenceT >
class Grain{
public:
        typedef SourceT source_type;
        typedef SourceEssenceT source_essence_type;
        typedef GrainEnvelopeT envelope_type;
        typedef GrainEnvelopeEssenceT envelope_essence_type;
        ...
private:
    source_type source_;
    envelope_type envelope_;
};


template< class SequenceStrategyT, class GrainT >
class Scheduler{
public:
        SequenceStrategyT sequenceStrategy;

        GrainT::source_essence_type sourceEssence;
        GrainT::envelope_essence_type envelopeEssence;
        ...
private:
        grain_type *grainPool;
};
```

An implementation of a Stochastic Delay Line Granulator using a stochastic density SequenceStrategy and a parabolic Envelope would look like this:

```
class StochasticDelayLineGranulator{
        typedef Grain< DelayLineSource< StochasticDelayLineSourceEssence >,
                        StochasticDelayLineSourceEssence,
                        ParabolicEnvelope< StochasticAmplitudeEssence >,
                        StochasticAmplitudeEssence > grain_type;

        typedef Scheduler< StochasticDensitySequenceStrategy, grain_type > scheduler_type;

        scheduler_type scheduler;
        DelayLine delayLine;
public:
        DelayLineGranulator(int maxGrains, int maxDelayTime)
                : scheduler( maxGrains )
                , delayLine( maxDelayTime )
        { scheduler.sourceEssence.setDelayLine( delayLine ); }

        float synthesize( float input ){
                float result = scheduler.synthesize();
                delayLine.write( input + result * .2 ); // 20% feedback
                return result;
        }

        //  additional facade methods for accessing synthesis parameters
        //  in scheduler.sequenceStrategy, scheduler.envelopeEssence and
        //  scheduler.sourceEssence go here
};
```

Note that `DelayLineSource<>` and `ParabolicEnvelope<>` are parameterized by Essence types. This reflects the abstract nature of Envelope::Essence and Source::Essences as described in the previous section, and allows other (perhaps non-stochastic) Essences to be used in combination with the existing implementations of DelayLineSource and ParabolicEnvelope.

Without the benefits of compile-time specialization as provided by language facilities such as C++ templates, this type of abstraction would not be practical for constructing an efficient real-time Granulator.

Optimizations

A common approach to improving the performance of software synthesis algorithms is to synthesize a vector (array) of samples at a time. This allows synthesis state variables to reside in registers or Level 1 cache for the duration of the vector computation, reduces the number of function calls per generated sample, and provides optimizing compilers with opportunities to unroll loops.

Vectorizing Grain Scheduling

As grain scheduling must in general be sample accurate, the scheduling method must accommodate activation of grains at any time, thus precluding scheduling only on fixed vector boundaries. A simple way of accomplishing sample accurate scheduling is to use variable length vectors expressed as a pointer to their first element and an integer representing the vector's length. The implementation of `Scheduler::synthesize()` below extends the previously described per-sample synthesis function to accommodate both vectorized synthesis and sample-accurate grain scheduling:

```
void Scheduler::synthesize( float *out, int length ){

        synthesizeActiveGrains( out, length );

        while( nextOnset < length ){
                Grain& grain = activateGrain( sequenceStrategy.nextDuration() );
                grain.synthesize( out + nextOnset, length – nextOnset );
                nextOnset += sequenceStrategy.nextInteronset();
```

```
        }
        nextOnset -= length;
}
```

Handling the case where `activateGrain()` fails to locate an available grain has been omitted for clarity. Note that all active grains are synthesized prior to activating new grains, this allows grains that complete during the current vector to be re-activated immediately if necessary. A more efficient implementation might interleave synthesis and activation as it iterates through the grain pool.

Hoisting Grain Synthesis Boundary Conditions

Many software synthesis primitives may be split into two parts: First, compute a resultant sample (with possible state variable updates) and second, check for deterministic boundary conditions such as circular buffer wrap-around or envelope state change. These boundary conditions will be true for only a small fraction of generated samples, however when computing audio one sample at a time they must be checked for each sample generated. In a vectorized implementation it may be more efficient to hoist boundary checks outside inner loops provided that the inner loops can be guaranteed to halt before the boundary conditions occur:

```
void Grain::synthesize( float *output, int length ) {
        int remaining = length;
        do{
                int nextBoundary =
                 envelope_.nextBoundary( source_.nextBoundary( remaining ) );

                float *end = output + nextBoundary;
                do{
                        *output++ += source_.synthesize( envelope_.synthesize() );
                }while( output < end );

                envelope_.checkBoundary( nextBoundary );
                source_.checkBoundary ( nextBoundary );
                remaining -= nextBoundary;

        }while( remaining > 0 && !envelope_.atEnd() );
}
```

This requires Source and Envelope to implement their synthesis algorithms as three separate functions:

- `nextBoundary( maximum )` returns the lesser of `maximum` and the number of samples until the next boundary condition. Algorithms with no boundary condition such as the Parabolic Envelope algorithm presented earlier can simply return `maximum`.
- `synthesize()` calculates the next sample and updates any internal state, but does not check for boundary conditions.
- `checkBoundary( samplesPassed )` notes the number of samples which have passed and performs boundary checks and state transitions (such as envelope segment state transitions) as necessary.

This technique is only applicable when the overhead of computing `nextBoundary()` is low: often `nextBoundary()` can be implemented by simply returning the value of an internal counter.

Final Remarks

This article has been primarily concerned with defining an architecture that facilitates simple and efficient implementation of many of the known variants of Granular Synthesis. It has made only passing reference to some significant forms of Granular Synthesis such as Pitch Synchronous Granular Synthesis and related techniques such as FOF synthesis, which can also be efficiently implemented using this architecture. Discussion of generating grain parameters has been limited to stochastic techniques, although many other possibilities exist. The treatment of Synthetic Grain Granular synthesis has been minimal. The fact that these topics fall outside the scope of this exposition reflects the breadth and depth of the granular concept.

The separation of concerns between SequenceStrategy, Scheduler and Grain can be considered a computer music design pattern with applications beyond Granular Synthesis. It is hoped that this article has presented a new angle on an old idea and may inspire others to explore some of its latent possibilities hitherto unheard.

Acknowledgments

I owe a debt of gratitude to Curtis Roads for his pioneering work in the field of Granular Synthesis and for introducing me to a number of its variants. Many thanks to Barry Truax for his 1988 *Computer Music Journal* article which started me on my journey of implementing real-time granular synthesis. Thanks to Gordon Monroe for providing me with the density to interonset time function presented in this article. Thanks are also due to Dean Walliss and Ken Greenbaum for their kind assistance and comments during the preparation of this manuscript.

References

1. Curtis Roads, "Automated Granular Synthesis of Sound," *Computer Music Journal* 2(2) (1978): 61-62. Reprinted in *Foundations of Computer Music*, eds. C. Roads and J. Strawn (Cambridge, Massachusetts: The MIT Press, 1985), 145 – 159. The 1985 version revises and expands Curtis Roads' seminal article on Granular Synthesis. The theory of granular synthesis is introduced in the context of Gabor's theory of acoustic quanta, and Xenakis' compositional theory of sound grains. Two implementations of Granular Synthesis and their compositional applications are discussed.

2. Barry Truax, "Real-Time Granular Synthesis with a Digital Signal Processor," *Computer Music Journal* 12(2) (1988): 14-26. This article describes an implementation of real time granular synthesis for the DMX-1000 signal processor. The article makes the useful distinction between delay line, sampled sound and synthetic grain granular synthesis that is used here. A system for varying synthesis parameters over time is also presented.

3. Barry Truax, "Discovering Inner Complexity: Time-shifting and Transposition with a Real-Time Granulation Technique," *Computer Music Journal* 18(2) (1994): 38-48. This article discusses the application of granular synthesis to pitch and time based transformations.

4. Curtis Roads, *the computer music tutorial*, (Cambridge, Massachusetts: MIT Press, 1996), 168 – 184. This unparalleled survey of the computer music field presents an extensive survey of granular synthesis techniques and an analysis of their application to various musical tasks.

5. Sergio Cavaliere and Aldo Piccialli, "Granular Synthesis of Musical Signals," in: *Musical Signal Processing,*eds. Curtis Roads et al., (Lisse: Swets & Zeitlinger, 1997), 155 – 186. This article describes an analysis/resynthesis architecture based on Granular Synthesis. An extensive analysis of the spectral properties of granular signals is presented.

6. Rodney Waschka II and Toze Ferreira, "Rapid Event Deployment in a MIDI Environment," *Interface, a journal of new music research* 17(4) (1988): 211-222. This article describes an implementation of the granular concept using an Atari 1040stf connected to a MIDI synthesizer.

7. Erich Gamma et al., *Design Patterns: Elements of reusable Object-Oriented Software,* (Reading, Massachusetts: Addison Wesley, 1995), 185 – 193. This book jump-started the design patterns movement – it is an excellent source of reusable Object Oriented design patterns. The "Facade" design pattern referred to here is concerned with providing an external interface to a set of collaborating objects in order to simplify or hide their internal details.

8. Truax, “Real-Time Granular Synthesis with a Digital Signal Processor,” 14.

9. Curtis Roads, *Computer Music Workshop at the Next Wave Festival*, (Unpublished Lecture, Melbourne, 1998). At this lecture Curtis Roads presented an extensive exposition of Granular Synthesis concepts. Included were discussions of grains with internal glissandi and the application of independent bandpass filtering to individual grains.

10. Curtis Roads, *the computer music tutorial*, 296 – 305.

11. James McCartney, “Synthesis without Lookup Tables,” *Computer Music Journal* 21(3) (1997): 6. This article describes a number of computationally efficient techniques for synthesizing envelope functions and periodic signals.

12. McCartney, 5.

13. Gordon Monroe, “A note on interonset times” (Private correspondence, 1998), 1. In this note Gordon Monroe provided the mathematical function approximating density to interonset time conversion used here. An exact, but more complex function was also described.

14. Ross Bencina, *AudioMulch Interactive Music Studio*, (Melbourne, 2001), software available from http://www.audiomulch.com/. This software implements a number of Granular Synthesis variants including Delay Line, Sampled Sound and Filtered Grain Delay Line Granular Synthesis. All of the granular synthesizers implement Grain Quantization as mentioned here.

15. Andy Carlson, “Essence,” in: *Pattern Languages of Program Design 4*, eds. N. Harrison, B. Foote and H. Rohnert (Reading, Massachusetts: Addison Wesley, 2000), 33 – 40. This article describes the “Essence” design pattern, which involves using a separate class to encapsulate the initialization parameters for one or more other classes.

16. Matthew H. Austern, *Generic Programming and the STL*, (Reading, Massachusetts: Addison-Wesley, 1998), 16 – 19. This book is a guide to applying generic programming techniques to C++, within this context it presents a complete reference to the C++ Standard Template Library.