GranCloud - A New SuperCollider Class for Real-time Granular Synthesis

Terry A. Lee
Masters Candidate, College of Music, University of North Texas
tlee@tleestudio.com, www.tleestudio.com

Abstract

GranCloud is a new SuperCollider class for the generation of real-time granular synthesis. GranCloud objects represent granular clouds and encapsulate all of the control data and methods necessary to render virtually any paradigm of granular synthesis using one simple interface. The class was designed to be flexible and extendable. Users may select from presets or to define their own grain synth definitions and control parameters. Data for each parameter may be stored as static values or objects that change value over time, including: functions, envelopes, streams, bus mappings, or any object that returns numeric values. The GranCloud class pushes the repetitive work involved in managing control data, grain scheduling, and grain parameter calculations "behind the scenes," allowing composers to focus more on what is most important to their compositions: the music.

1 Introduction

Granular synthesis is a powerful and flexible approach to synthesis capable of generating a wide variety of rich sonic textures. Generating granular synthesis, however, can be a tedious and cumbersome task, especially when done in real-time. There is a tremendous amount of control data to manage. The work involved in tracking the data, making grain specific calculations, and scheduling grains can slow down the compositional process significantly.

GranCloud is a SuperCollider class that was designed to help simplify the process of generating granular synthesis. GranCloud objects encapsulate all of the control data and methods required to generate the granular clouds they represent. The encapsulation simplifies the composer's task by removing the need for complex programming to perform grain parameter calculations and grain scheduling within individual compositions. GranCloud also provides a simple interface for the composer to define and manipulate the control parameters of the cloud.

2 GranCloud

The GranCloud class defines SuperCollider objects representing instances of granular clouds. Like the SuperCollider application, GranCloud is open-source and free for anyone to download and use. It is available for download at www.tleestudio.com/grancloud. The distribution includes the GranCloud class definition, documentation, examples, and several presets synth definitions for various types of granular synthesis. An alpha version of GranCloudGUI is also available for download. GranCloudGUI is a powerful graphic interface, still under construction, for manipulating and shaping granular clouds using the GranCloud class. Questions and comments may be sent to the author at tlee@tleestudio.com.

2.1 The Class Definition

The GranCloud class definition is a file containing code that is compiled into the SuperCollider class library when the program is started up or the library is recompiled. Once obtained, the file is placed in the SCClassLibrary directory of the SuperCollider installation. After the library has been recompiled, or the application restarted, GranCloud objects may be created and used in any application run by that SuperCollider installation.

The class definition contains code defining a data structure for storing all the control data needed to represent granular clouds and the programatic methods used to render clouds in real-time. Since the code needed to perform grain parameter calculations and scheduling is contained in the class definition, the composers do not need to duplicate the code within their own compositions to render granular synthesis. Instead, they simply choose or define a grain synth definition, initialize a GranCloud object, set the control data, and instruct the cloud to "play" itself at the appropriate time.

2.2 GranCloud Objects

Instances of GranCloud objects may be created by sending the *new* or *preset* message to the GranCloud class within any SuperCollider program. The GranCloud object returned can be stored in a variable for easy reference later in the application. After initialization, a grain synth definition is specified and various control parameter values are stored by name in instance variables within the object. With the control data defined, the cloud object knows it's own shape, duration, how it changes over time, how the grains are organized, and how to render itself. Every piece of information and programming needed to generate the synthesis is encapsulated within the object.

2.3 GranCloud Design

GranCloud was designed to be both simple to use and flexible enough to handle any type of granular synthesis imaginable. Since there are many different types of granular synthesis and many ways that grains may be structured and organized in time and space, simple methods to maximize flexibility and extendibility were important to the design.

Some of the goals of development include: (1) the object should be able to generate any form of granular synthesis; (2) clouds should be able to morph seamlessly between different states and types of granular synthesis; (3) any sound source should be able to be used for the grain, including live input, buffered samples, and synthesis; (4) any envelope shape should be able to be used and altered from grain to grain; (5) any type of signal processing should be able to be used and altered with each grain; (6) redundant code to perform grain parameter calculations and scheduling should not be required in every application; (7) user-defined grain parameters should be able to be added as needed, and (8) control data that changes over time should be able to be represented in many different ways.

Accomplishing these goals required careful design, separating the process into three parts: a synth definition defining the internal structure of the grain, a control data repository where different types of control data are stored by parameter name, and a rendering engine that calculates specific parameter values based on the control data and schedules each grain synth node on the server.

The Grain Synth Definition A SuperCollider synth definition (SynthDef) is used to define the internal structure of each type of grain. A grain SynthDef is a synthesis template for a single grain of any type. The SynthDef tells the SuperCollider server what signal generators to hook together to generate the audio signal for a single grain. Since users can define their own SynthDefs, any sound source may be used, any envelope applied, and any type of signal processing may be done within the grain. The synthesis that occurs within the grain may be as simple or as complex as the user wishes.

Each SynthDef has named parameters allowing numeric data to be passed in when each grain is created. The parameter values can be used to control characteristics of the grain, such as frequency, duration, amplitude, sound source, envelope shape, and amount of processing. The user may define any parameter names they wish as long as the same names are used when control data is set for the parameters in the GranCloud object. Any number of parameters needed to support the synth node may be used.

The only requirement of a SynthDef used with GranCloud is that it knows how to "free" itself from the server once the grain is done processing. More information on SynthDef design and how they can be set to free themselves can be found in the SuperCollider help files.

For users not familiar with SuperCollider or SynthDef design, several preset SynthDefs have been built into the GranCloud class. The presets may be used by creating GranCloud objects using the *preset* class method, passing the chosen preset name in as an argument.

Control Data There are three ways that control data for grain parameters may be stored in GranCloud objects, depending on the type of data.

Static data is stored in an array of parameter name/value pairs in the *args* instance variable of the GranCloud object. The array is delivered "as is" to the grain synth node. Values in the array may be changed while the cloud is rendering to change the state of the grain, but generally the data does not change from grain to grain.

The second method stores dynamic data for parameter values that change from grain to grain. Since there are several different ways that dynamic data may be represented, GranCloud was developed to accept many different types of data objects for dynamic data. These objects include: static numbers, user-defined functions, SuperCollider envelope (Env) objects, SuperCollider stream or pattern objects wrapped in streams, an array of explicit values converted to a stream, or any user-defined object that responds with a number when it receives the *next* or *value* message.

Dynamic control data is set by parameter name in three Dictionary objects that are stored in the *center*, *dev*, and *dist* instance variables within the GranCloud object. The Dictionary objects keep track of both the name and value for each parameter as it is set. The three dictionaries store center values, optional deviation values, and optional distribution functions for each parameter by name. More information on the purpose of each data type follows in subsequent sections. The same names used for parameters in the grain SynthDef should be used when storing data in the dictionaries.

The third way grain parameter control data may be specified is through bus mappings. Bus mappings are stored as an array of parameter name/bus index pairs in the *map* instance variable. When each grain is scheduled, the named parameters in the array will be read from the associated control bus. The bus values can be controlled by external processes. This allows grain parameters to be mapped to external sources such as signals from other synth nodes, OSC Messages from other applications, MIDI Controllers, HID devices, or other homemade or commercial controllers.

Parameter Randomness It is very common for granular synthesis parameters to be governed from grain to grain by a random or pseudo-random process. The storage of dynamic data was broken into the center, deviation, and distribution function components in order to support parameter randomness. If the deviation value is provided, the distribution function is executed with the deviation value as a parameter and the result added to the grain center value.

The random function used to generate the actual deviation for each grain is also user-definable. Any of the built-in SuperCollider random functions that operate off of a single value may be used, or the user may define their own function that receives the deviation value as a parameter. This way any form of distribution may be used, including linear, exponential, Gaussian, or other types of random or patterned distributions.

If a deviation value is set but the distribution function is not, the distribution function defaults to the SuperCollider *rand2* function which returns linear random distribution over the deviation range.

The Rendering Engine The primary job of the rendering engine is to schedule grain synth nodes on the server while delivering the appropriate parameter values based on the control data. The engine is built into the *play* method of the GranCloud class. When called, the *play* method performs all of the parameter calculations needed for each grain and sends a message to the SuperCollider server to schedule the grain synth node at the appropriate time.

The timing of grains is controlled by a dynamic parameter named *rate* that must be set in the control data. The *rate* parameter is the only parameter that is absolutely required by GranCloud. Although the rendering engine does not need to know the purpose of any of the other grain parameters, it does need to know which parameter controls the scheduling of each grain.

The *rate* parameter represents the grain rate in seconds, which for GranCloud means the amount of time between the start of each grain. The *rate* parameter is completely independent of the grain duration, so multiple grains may or may not overlap depending on the parameter values. Since the grains know how to "free" themselves, all the scheduler needs to know is when to start the each grain.

Control data for the *rate* parameter is stored and processed in the same manner as other dynamic parameters, so it may return static or changing data with each grain depending on the data types used to represent it. This allows for any type of synchronous, asynchronous, or patterned granular synthesis to be generated using the same simple interface.

As the rendering engine is scheduling each grain it must convert the stored control data into discrete values for each grain. When each grain is scheduled, the rendering engine loops through each parameter set in the *center* dictionary and calculates a discrete center value based on the data and the play time of the cloud. Then it checks to see if a deviation value is set for that parameter. If so, it calculates a deviation value specific to the play time. Then it executes the distribution function, using the deviation value as parameter, to get an actual deviation value. The actual deviation is added to the center value to get the actual value that is delivered to the synth node for that grain. Formula 1 summarizes the calculation.

$$value = center + f_d(deviation)$$

Formula 1: Parameter Value Calculation

Once all the discrete parameter values have been calculated, they are added to the static parameter array stored in the *args* instance variable and delivered to the server with the SynthDef name and grain start time. An accompanying message to the server maps any parameters to buses as specified in the *map* instance variable array. Once the grain node has completed execution, it frees itself from the server.

The cloud will render until the time duration specified in the *duration* instance variable has past, or the *stop* message has been sent to the GranCloud object.

2.4 Applications of GranCloud

GranCloud is flexible enough that virtually any paradigm of granular synthesis may be generated by it. It is impossible to list all of the different ways in which it might be used. The author has used it for soundfile granulation, pitch shifting, time stretching, granulation of a live input, harmonization, sustaining live input, generation of clouds based on synthetic grains of various types, and creating gestures that morph between different types of granular synthesis. He has experimented with various types of grain-by-grain processing and three-dimensional spatialization of grains. The only real limits to GranCloud are the imagination of the user and the speed of the hardware.

3 Using GranCloud

There are three basic parts of a GranCloud implementation: the SynthDef which defines the structure of the grain, the control data stored by parameter name within the GranCloud object, and the rendering engine defined in the play method in the GranCloud class. Using GranCloud effectively requires a basic understanding of how each part works and how they relate.

3.1 The Synth Definition

The first step to using GranCloud is to define the SynthDefs that will be used to generate the grains.

Just as the building block of a granular cloud is the grain, the building block of a grain for GranCloud is a SynthDef. SynthDefs are templates for creating grain synth nodes on the the SuperCollider server. GranCloud expects that SynthDefs defining grain types will be sent to the server before the cloud is rendered. The SynthDefs tell the SuperCollider server specifically how to generate the audio signal for the grain.

Users may create their own SynthDefs or select one of several presets built into the GranCloud class. GranCloud objects using the preset SynthDefs may be created using the "preset" constructor method, passing in the Server object and preset name as arguments. When GranCloud objects are constructed using the preset method, the appropriate SynthDef is immediately sent to the server in preparation for synthesis.

User-defined SynthDefs should be sent to the server when the SynthDef is defined. The user-defined SynthDef name should then be stored in the "def" instance variable of the GranCloud object once it has been created. A detailed discussion on SuperCollider SynthDef design is beyond the scope of this paper, however there is good documentation in the SuperCollider help files.

A typical grain synth node will do five basic things. It will receive grain argument values into named parameters, generate or read a source sound, apply an envelope to the source, process the enveloped grain, and write the resulting signal to an audio bus. Since all of this is defined within the SynthDef, the GranCloud object does not need to know anything about the SynthDef internals except what parameter values to pass into the node. Any type of signal generation and processing may be done within the grain.

Grain arguments are specified at the top of each SynthDef by parameter name. Any names for parameters may be used as long as they correspond with the names used when setting the control data. Normally a parameter named *out* is specified to tell the grain the bus index that should receive the audio signal.

The source audio signal can originate from any source. It may be read from a sample buffer or audio bus, or may be synthesized within the grain. It may originate from a live input or signal generated from an external process.

Any envelope shape may be used, or not used. Typically an envelope shape is defined using a SuperCollider envelope (Env) object and converted to an audio or control rate signal using an envelope generator (EnvGen). The envelope signal may then be multiplied to the source signal before or after additional processing is applied. Grain parameters may be used to specify and change the shape of the envelope from grain to grain. The EnvGen unit generator may also be used to free the synth node from the server by setting the doneAction parameter of the EnvGen to 2.

Virtually any type of processing may be added to the signal before or after the source signal is enveloped. Examples may include applying grain-by-grain filtering, spatialization, panning, or reverb. User-defined parameters can supply any parameter value needed to support grain-by-grain processing.

The final task of the synth node is to write the processed signal out to an audio bus. Any number of channels of output may be written depending on the SynthDef design. It may write directly to a hardware output bus or an internal bus for further processing of the cloud as a whole. For convenience, the grain scheduler will provide the output bus number to use for the grains to the *out* parameter. The *out* bus index may be specified when the rendering engine is instructed to play. If undefined, it defaults to the first hardware output bus of the system.

Example 1 demonstrates the creation of a simple grain SynthDef. The SynthDef is named "sine_grain" and has five grain arguments specified. The sound source is a simple sign wave generator multiplied by a psuedo-gaussian envelope signal. The EnvGen unit generator has been given a doneAction of 2, so it will automatically free the synth node when the envelope has completed. The resulting signal is then processed with a simple panning unit generator and the resulting two channel signal written to the bus specified in the *out* parameter.

```
// boot the local server
s = Server.local.boot;

// define SynthDef named "sine_grain"
SynthDef("sine_grain", {
   // specify grain args
   arg out, freq, dur, amp, pan;
   // define variable for audio
   var signal;

   // generate source sound
   // and multiply by envelope
   signal = SinOsc.ar(freq, 0, amp) *
```

```
        EnvGen.kr(
            Env.sine(dur),
            doneAction: 2
        );

    // process with panning ugen
    signal = Pan2.ar(signal, pan);

    // write to out bus
    Out.ar(out, signal);

}).send(s);
```

Example 1: A Simple Grain Synth Definition

3.2 Initializing the GranCloud Object

Once the grain SynthDef has been defined or a preset chosen, the next step is to create an instance of a GranCloud object. This is done using one of the two constructor methods in the GranCloud class.

The default constructor method is the *new* method, which should be used if a user-defined SynthDef is to be used (see Example 2). Once initialized the object can be stored in a variable for later reference. The grain SynthDef name is then stored in the *def* instance variable within the object.

```
// initialize a new object
a = GranCloud.new;

// specify the SynthDef by name
a.def = "sine_grain";
```

Example 2: Constructing a GranCloud object using the new class method

If a preset is to be used, the *preset* class method should be called to construct the GranCloud object (see Example 3). The *preset* method requires two parameters: a Server object and the name of the preset. Available presets are listed in the GranCloud help file included in the distribution. When the *preset* method is called, GranCloud automatically sends the appropriate SynthDef to the server and populates the *def* instance variable with the SynthDef name. In addition, default values set for all grain control parameters.

```
// boot the server
s = Server.local.boot;

// initialize a preset
a = GranCloud.preset(s, "buf_grain");
```

Example 3: Constructing a GranCloud object from a preset

Once the GranCloud object has been created the play duration of the cloud may be set in the duration instance variable (see Example 4). The play duration defaults to “inf” or infinity, meaning that once told to play, the cloud will render until explicitly told to stop. The play duration may also be set to a number, representing the number of seconds the cloud will render before stopping itself.

```
// set the duration in seconds
a.duration = 100;
```

Example 4: Setting the play duration of the cloud

3.3 Setting the Grain Control Data

The next step to using GranCloud is to set the grain parameter control data for the cloud. The grain parameter control data determines the shape of the cloud. There are three basic types of control data that may be used for each grain parameter: static data that is the same for every grain, dynamic data that may be different for every grain, and external data that is read from a control bus.

Static Data Static data may be stored in an Array in the GranCloud instance variable named *args*. The array contains parameter name/value pairs that will be delivered verbatim to every grain synth node. Static parameter values may be changed while the cloud is rendering by changing the values in the array, but generally they remain static through the life of the cloud. Example 5 demonstrates how to store static data in the *args* instance variable.

```
// set name/value pairs for parameters
// that do not change grain to grain
a.args = [  'amp', 0.2, 'dur', 0.05 ];
```

Example 5: Setting Static Control Data

Dynamic Data Dynamic data is set in the three dictionaries stored in the *center*, *dev*, and *dist* instance variables as discussed above. Center and deviation values may be stored as numbers, functions that return numbers each time they are evaluated, envelope objects representing a shape that changes continuously over time, streams that return a numeric value every time a *next* message is called on the object, or any SuperCollider object that returns a numeric value when a *next* or *value* message is called.

If a deviation value is set for a parameter, then a distribution function may be set as well. Distribution functions can either be stored as a name to a built-in SuperCollider distribution function or an actual user-defined function. If no distribution function is specified, the built-in *rand2* function is used, defining a linear random distribution.

It must be remembered that a value for the *rate* parameter, defining the grain rate, is required for the GranCloud object to render correctly. The *rate* parameter should be set using the *center*, *dev*, and *dist* dictionaries, the same as any dynamic parmeter.

Example 6 demonstrates several methods of setting dynamic parameter values using different data types.

```
// define random grain rate
// using deviation and distribution
// function
a.center[\rate] = 0.01;
a.dev[\rate] = 0.005;
a.dist[\rate] = \rand2;

// use a function to set grain duration
// equal to be 3 times the
// calculated center value
a.center[\dur]={arg g; g[\center] * 3};

// crescendo from 0.1 to 0.4 over 5 sec
// using an envelope object
a.center[\amp] = Env.new(
    [0.1, 0.4],
    [ 5 ]
);

// choose center frequency from a
// sequential stream and add some
// deviation using a bell-like curve
a.center[\freq] = Pseq.new(
     #[220, 822, 567],
      inf
).asStream
a.dev[\freq]  = 20;
a.dist[\freq] = \sum3rand;
```

Example 6: Setting Static Control Data

3.4 Parameter Dependencies and Calculation Order

GranCloud includes support for setting dependencies between grain parameters. In the Example 6, the grain duration is set to be dependent on the grain rate using a function. When functions are evaluated for grain parameters, a data dictionary is passed into the function as an argument that contains the values already calculated for other grain parameters of the grain. Calculations can be made within the function based on the previously calculated values.

In order to use dependencies, a processing order for the parameters must be specified or dependent parameters may be executed before the parameters they depend upon. The order need only be specified if dependencies are used, and only the parameters that are involved in the dependency need to be ordered. Parameters not included in the order will be processed in an arbitrary order. Parameter order is set in an array of parameter names in the *order* instance variable in the GranCloud object (see Example 7). Circular dependencies are not allowed and may cause unexpected results.

```
// specify the processing order
// for dependent parameters
a.order = [ \rate, \dur ];
```

Example 7: Setting Static Control Data

3.5 Mapping Parameters to Control Busses

In some applications of GranCloud it may be desirable for parameter values to be mapped to a signal on a control bus. The signal may be written to the bus by any external process with an order of execution that precedes the cloud. This is done by setting parameter names and bus indexes in name/value pairs in an array stored in the *map* instance variable of the GranCloud object (see Example 8).

```
// map to a bus
a.map = [ \freq, 1, \amp, 2 ];
```

Example 8: Mapping Control Parameters to Buses

3.6 Rendering the Cloud

With the synth definition sent to the server, the GranCloud object initialized, and the control data set, the cloud may now be rendered. The cloud is rendered using the *play* method. Arguments to the play method include: the SC Server object, the output bus where the audio signal for grains should be written, the target SC group on the server in which the grains will be executed, an optional start time offset, and an add action for handling order of execution on the server. Defaults suitable for most applications are provided for each parameter, so usually only the server, output bus, and sometimes the target group needs to be provided.

When the play method is called, the cloud begins rendering immediately and will continue until the cloud duration is over or the stop method is called.

The stop method will stop the engine from scheduling any more grains. Any grains that have already been scheduled will finish execution so clicks are not heard because of grains being sliced in the middle. Example 9 demonstrates using the play method.

```
  // render the cloud
  a.play(s, 0);

  // stop the cloud
  a.stop;
```

Example 9: The Play and Stop Methods

3.7 Other Functionality

Other functionality not covered in this paper is described in the GranCloud help file. This includes, looping support for parameter values as well as the cloud as a whole, constraining parameter values to specified ranges or step sizes using ControlSpec objects, and controlling parameter values with graphical interfaces, MIDI controllers, and HID devices such as game controllers.

4 A Complete Example

The following example demonstrates the complete process of generating granular synthesis using GranCloud. In this example, a soundfile is loaded into a buffer and granulated. Both pitch shifting and time expansion is done as well as grain by grain filtering.

```
// boot the local server
s = Server.local.boot;

// define a synthdef for a buffer
// based grain and send to server
SynthDef(\bufgrain, {
  // name grain arguments
  arg out, buf, dur, amp,
      bufRate, startPos, pan,
      envCenter, ffreq, rq;

  // define some variables
  var env, sound;

  // define an evelope that
  // changes shape based on
  // the envCenter argument
  env = EnvGen.kr(
    Env.new(
      [ 0, amp, 0 ],
      [ dur * envCenter,
        dur * (1 - envCenter)
      ],
      'sine'
    ),
    doneAction: 2
  );

  // read source from buffer,
  // start position and read rate
  // are based on parameters
  sound = PlayBuf.ar(
   1, buf,
   BufRateScale.kr(buf) * bufRate,
   1,
   BufFrames.kr(buf) * startPos,
   0
  );

  // apply the envelope to the sound
  sound = sound * env;

  // apply some filtering on the sound
  sound = BPF.ar(sound, ffreq, rq);

  // apply a spatial position
  sound = Pan2.ar(sound, pan);

  // write the output to a bus
  Out.ar(out, sound);

}).send(s);

// reate the sound into the buffer
b = Buffer.read(s, "test_sound.aif");

// initialize the cloud
a = GranCloud.new;
a.def = \bufgrain;
a.duration = 10;

// set static control data
a.args = [ \buf, b.index, \rq, 0.2 ];

// set dynamic control data
a.center[\rate] = 0.01;
a.dev[\rate] = 0.002;
a.dist[\rate] = \sum3rand;

a.center[\dur] = {arg g; g[\rate] * 3 }

a.center[\bufRate] = Env([ 1, 2 ], 10);
a.dev[\bufRate] = 0.1;

a.center[\startPos] = Env([ 0, 1], 10);
a.dev[\startPos] = 0.002;

a.center[\amp] = Prand.new(
  #[0.1,0.2,0.3], inf
).asStream;

a.center[\envCenter]={ 0.25.rrand(0.75) };
```

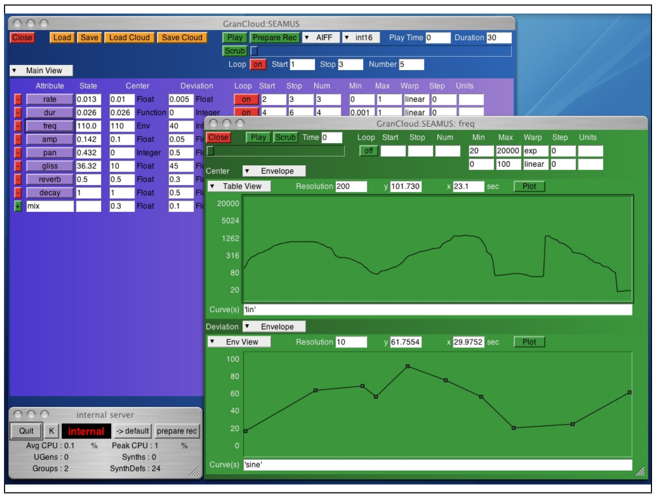
```
// map filter freq to control bus 1
a.map = [ \ffreq, 1 ];

// play the cloud
a.play(s, 0);

// stop the cloud
a.stop;
```

5 Looking Forward - GranCloudGUI

The author is currently developing a powerful graphic interface for building and manipulating granular clouds using GranCloud objects. An alpha version of the program is available for download with the GranCloud distribution. GranCloudGUI allows users to define and manipulate grain control data graphically in several different ways, including the ability to “draw” shapes or graphically manipulate envelopes or sliders. The graphic interface allows users to approach granular synthesis in a much more intuitive manner than coding. Once GranCloud objects are created and tweaked using the interface, they can be “saved” and recalled for use within any composition, or rendered directly to audio files.



Attribute	State	Center		Deviation		Loop	Start	Stop	Num	Min	Max	Warp	Step	Units
rate	0.013	0.01	Float	0.005	Float	on	2	3	3	0	1	linear	0	
dur	0.026	0.026	Function	0	Integer	on	4	6	4	0.001	1	linear	0	
freq	110.0	110	Env	40	Int	off				20	20000	exp	0	
amp	0.142	0.1	Float	0.05	Fl					0	100	linear	0	
pan	0.432	0	Integer	0.5	Fl									
gliss	36.32	10	Float	45	Fl									
reverb	0.5	0.5	Float	0.3	Fl									
decay	1	1	Float	0.5	Fl									
mix		0.3	Float	0.1	Fl									

Figure 1: GranCloudGUI: A Powerful Graphic Editor for GranCloud Objects

6 Conclusion

Granular synthesis is a powerful and flexible compositional paradigm. The sonic variations that can be achieved using the technique are nearly endless.

Bringing granular synthesis and real-time together can be a difficult marriage to manage, but with the appropriate tools, the job can be made much easier. GranCloud is one such tool. It is the result of one composer struggling with difficult technical issues and striving to create simple solutions.

It is with pleasure that the author shares GranCloud, the result of many hours of hard work and spousal frustration, developed in hopes of providing simpler solutions to the exciting realm of real-time granular synthesis. It is the music that is important. Why should technical details get in the way?

References

Boulanger, R. (2000). *The Csound Book*. Cambridge, Massachusetts: MIT Press.

Keller, D. and B. Truax (1998). Ecologically-based granular synthesis. In *Proceedings of the International Computer Music Conference*, pp. 117–120. International Computer Music Association.

McCartney, J. (1996). *SuperCollider, A Real-time Sound Synthesis Programming Language*. Austin: AudioSynth.

McCartney, J. (1998). Continued evolution of the supercollider real time synthesis environment. In *Proceedings of the International Computer Music Conference*, pp. 133–136. International Computer Music Association.

Roads, C. (2001). *Microsound*. Cambridge, Massachusetts: MIT Press.

Truax, B. (1994). Discovering inner complexity: time-shifting and transposition with a real-time granulation technique. *Computer Music Journal 18*(2), 38–48.