A Max Patch to Generate a Single Grain

The technique for producing a single grain is essentially very straightforward. A section of a sound file is multiplied by a windowing function to produce an enveloped burst of sound at a predetermined point and of a predetermined length. In his original works, Gabor used a Gaussian distribution curve to envelope the sound although many other windows are possible and even preferable, as the Gaussian curve will not produce as much amplitude as, say, a Hann window. For this reason some alternative windowing functions will also be described in a later section.

The Gaussian Window Function

A Gaussian windowing function produces a bell shaped curve, which can be expressed as follows:

$$\omega(n) = e^{-\frac{1}{2}(\frac{n-(N-1)/2}{\sigma(N-1)/2})^2}$$

$$\sigma \leq 0.5$$

Where N represents the width in samples of the window, n is an integer with values

$0 \leq n \leq N-1$ and σ is the standard deviation, which must be less than or equal to 0.5 in order to avoid discontinuities at the region boundaries.[1]

Max includes an expression object (*expr*) to evaluate mathematical functions such as this. The Gaussian distribution curve shown above can be generated using the following syntax inside the *expr* object:

exp(-.5*pow(($f1-($f2-1)/2)/(($f3* ($f2-1))/2)\,2))

Where exp is e, $f1 is n, $f2 is N, and $f3 is σ (which must be less than 0.5)

A Max MSP Patch to Generate a Gaussian Distribution Curve

Figure 1 shows an annotated patch, which generates a Gaussian distribution curve. At the heart of the patch is the *expr* object described above which calculates values for the Gaussian window. The *expr* has three inputs, one each for $f1, $f2, and $f3. (In Max the $ sign is used as a placeholder for a variable argument and the f stands for floating point number). On loading the patch, a bang is sent to the 1024 message, which sets the window size, N, to 1024 or 2^{10} samples. The value is passed to the second inlet of expr where it is substituted for $f2. The 1024 message also provides an argument for the *uzi* object, which sends out a series of bangs the number of which depends on its argument (1024 in this case). (In Max a bang is an empty message designed to trigger an output from an object or message). The *counter* object counts bangs at its left input before outputting its current count value at its left output. The arguments of the *counter* object set direction (0 is

[1] The standard deviation value is one of the most important factors in determining the final quality of the sound with this window. It is worth taking some time to experiment with this value as, if it is set too high or too low, there will be unwanted artifacts in the final sound. There is no universal setting for this but a value of 0.27 seems to be a reliable compromise between abrupt discontinuities at if the value is set too high and insufficient frequency information if the value is set too low.

forward) min count value (0) and maximum count value (1024 in this case). The value from the left output of *counter* is routed by the *t* (trigger) object to the *peek~* object and to the first input of *expr*, where it is substituted for $f1 (n – a value from 0 to 1023). The *peek~* object writes values into the buffer, treating it as a floating-point table object. The first input to *peek~* is the sample index, and the second input is the sample value, which in this case is the output of the *expr* object, i.e. the value of the Gaussian function at the particular sample index.

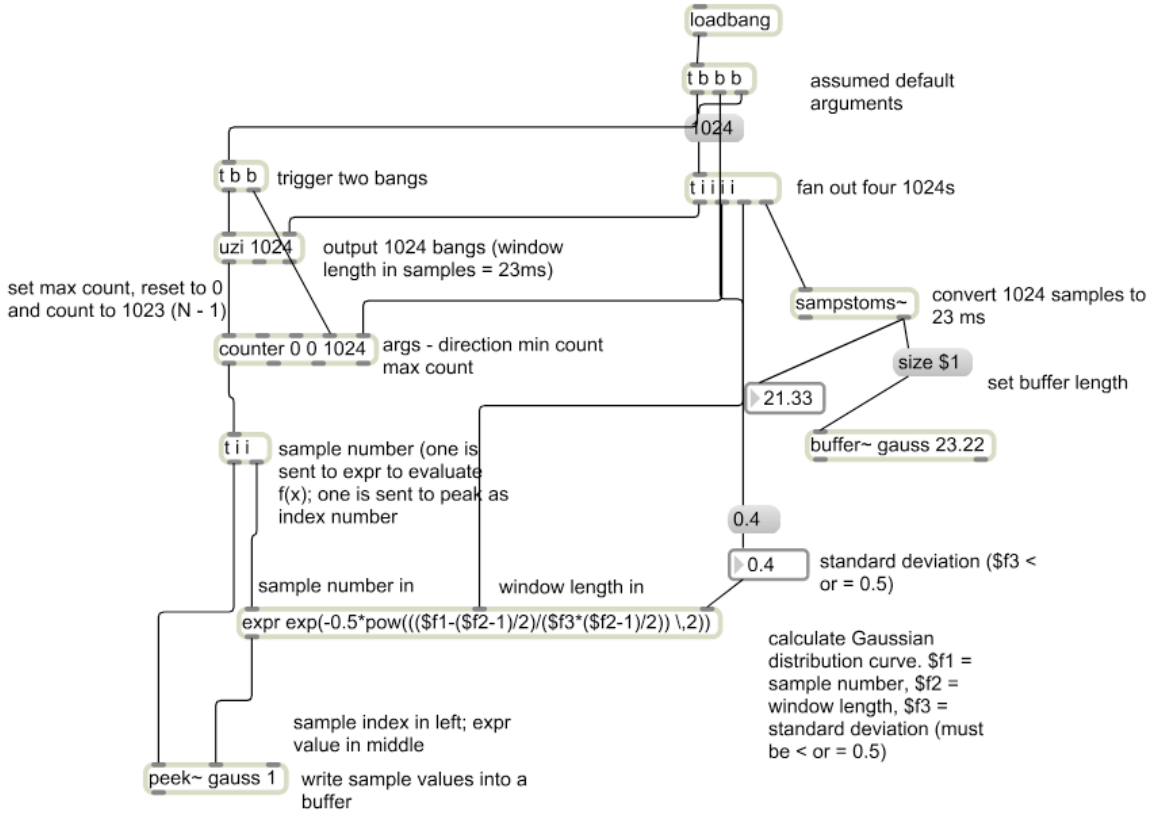

Figure 1: a patch to generate a Gaussian distribution curve

The value for σ (standard deviation) is also set on loading, (initialised as 0.4 in this case, although this can, of course, be changed).

The final output of the patch is stored inside the *buffer~* object. It can then be multiplied by a sound file section in order to produce a grain enveloped by the bell shaped curve described by a Gaussian function. The Gaussian curve inside the buffer~ is shown below in Figure 2:

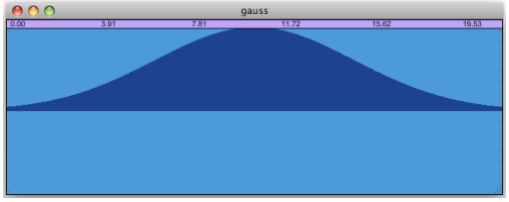

Figure 2: Gaussian distribution curve

The length of the window is shown on the horizontal axis in milliseconds (1024 samples at 44.1kHz in this case = 21.22 ms) and amplitude (on a scale from 0 to 1.) is shown on the vertical axis.

A Max MSP Patch to Generate One Grain

The next step is to construct a patch to generate a single grain by multiplying a section of a sound file (or any other appropriate signal) by the values in the Gaussian window. One possible patch to do this is show in Figure 3 below:

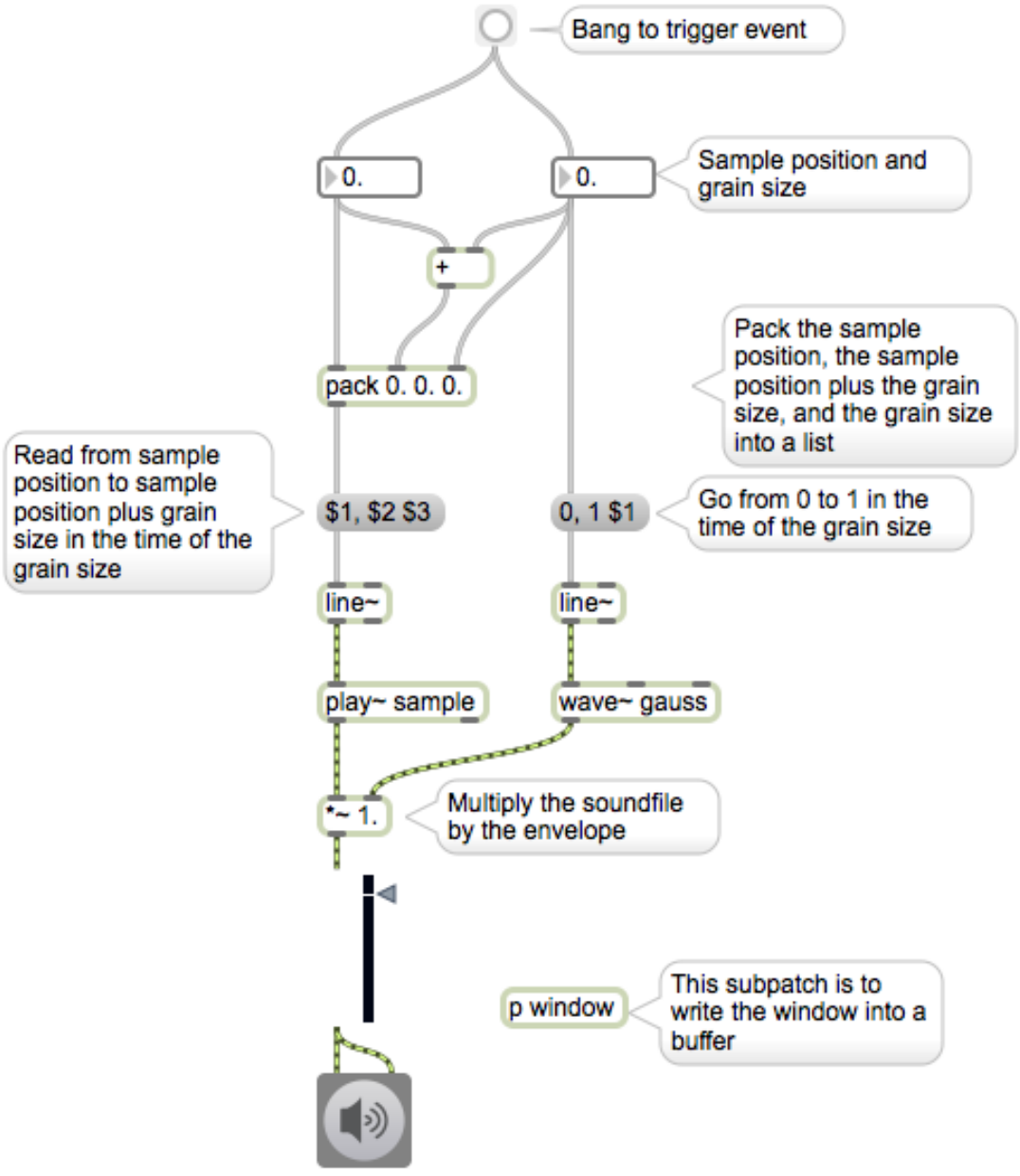

Figure 3: A patch to generate one grain

At the heart of this patch is the multiplication of the outputs of the *play~* object and the *wave~* object. The Gaussian distribution function is now embedded in a subpatch (p window). The *wave~* object uses the *buffer~* from the subpatch as a wavetable to read back

the values of the window. The *play~* object provides the sound file (also read from a buffer) which will be the source material for the grains.

Both the *play~* and the *wave~* objects are driven by *line~* objects, which output a ramp from one value to another over a specified period in milliseconds. The variable for the *line~* object driving *wave~* is the window duration, expressed inside a message box as: 0, 1 $1 (go from 0 to one in $1 milliseconds). $1 is substituted for whatever value is given as the grain size in milliseconds. The *line~* object driving the *play~* object takes three parameters to specify grain location and duration. The position in the sound file from which to start the grain is set in the number box at the top of the patch and this value is then added to the grain size to give the window duration. The time value for the *line~* object is also given by the grain size. These three values are then sent as a list (using the *pack* object) to the *line~* object, which interprets the list as follows: go from the *grain location* to the *grain location plus the window duration* in the time of the *window duration*.

The patch described above is an effective way of outputting a windowed burst of acoustic energy, a grain, of a specified duration and location within a sound file. The next step in developing an effective granular synthesis environment is to devise methods for triggering and controlling a great many grains repeatedly to produce streams or clouds of possibly many thousands of grains.

Max includes an object known as *poly~*. This provides a convenient way of embedding multiple instances of a patch within another patch in order to produce polyphony in a computationally efficient manner. It is possible to embed the patch described above in a *poly~* object in order to send control data from a higher-level patch. However, the patch will have to be modified in order to receive the control information and in order to function efficiently within the *poly~* environment.

Figure 4 shows one possible solution to adapting the patch for use within *poly~*. Once adapted, the patch in Figure 4 can then be loaded as an abstraction within a *poly~* object and any control values can be entered via the parent patch. A possible result is shown in Figure 5 where the metro object is used to trigger the grains. The metro object outputs a bang at a regular, user defined interval operating on the Max MSP control rate. This highlights a significant shortcoming with this design; namely that the metro object will always be limited by the rate of the Max scheduler. It will also be influenced by the global audio interrupt and scheduler settings for the program. If the scheduler speed is 20ms, for example, the grains cannot be denser that 1000 over 20ms (i.e. 50 grains per second). Also the grains will be quantized to a multiple of 20ms (ignoring any interrupts which will make this timing figure unreliable anyway). *Metro* is therefore not always a suitable object to use in an application such as this where very short durations and high grain densities are involved and accurate timing is essential.

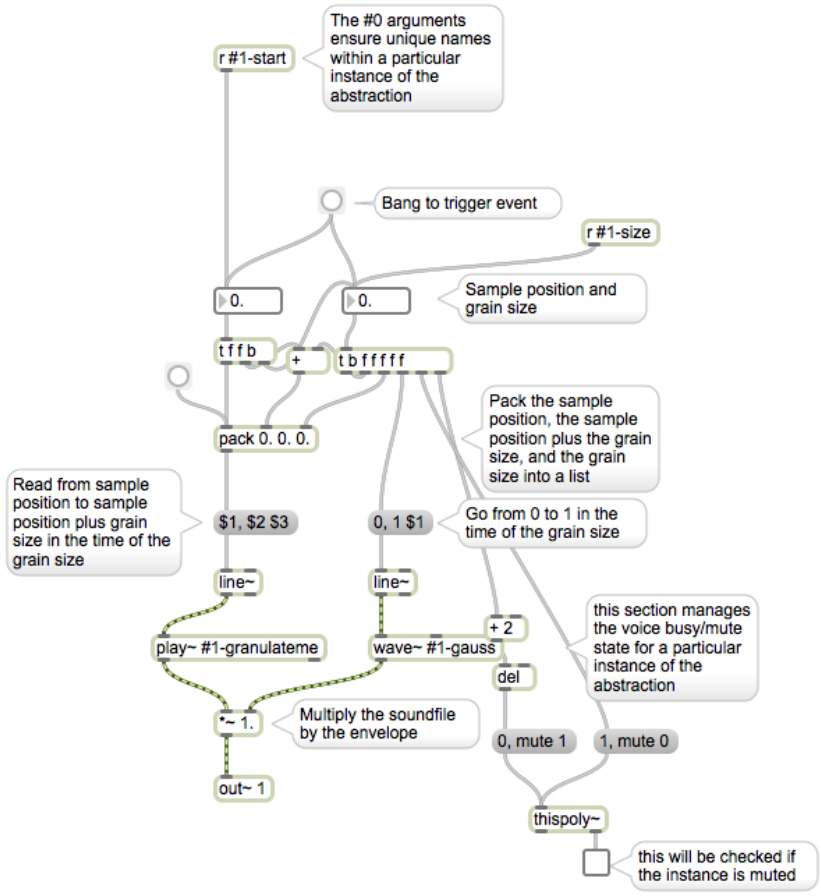

Figure 4: a patch for generating a grain adapted for use within the poly~ object

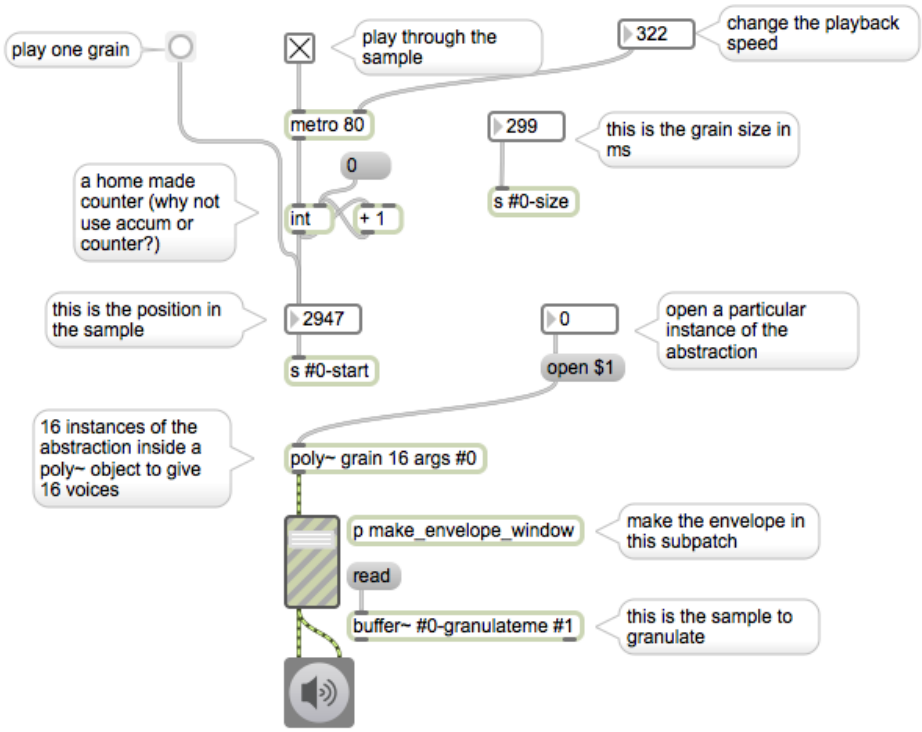

Figure 5: Embedding the grain engine in a poly~ object

For more reliable timing it is necessary to establishing a sample accurate method of triggering a stream of grains of a set duration and at a specified location. That is to say, a system is required which functions at the audio signal rate and is therefore not constrained by the rate of the Max scheduler. [2]

Figure 6 shows a patch adapted to play a stream of grains at the audio rate. The metro object has been replaced by a phasor~ object. The *phasor~* outputs a sawtooth wave which ramps from 0 to 1 at a specified frequency. It can be used as a sample accurate control signal for other objects, in the case the play~ and wave~. The output of the phasor~ is multiplied by the grain size and then offset by the start time of the waveform selection. In this way the sound file inside the *play~* object is triggered at the sample offset time and read for the length of the grain size. The frequency of the *phasor~* object in Hertz is calculated with reference to the grain size.

The output of the phasor~ is also used to trigger sample and hold objects in order to ensure that values are only passed on at the end of phasor~ ramp thus avoiding discontinuities.

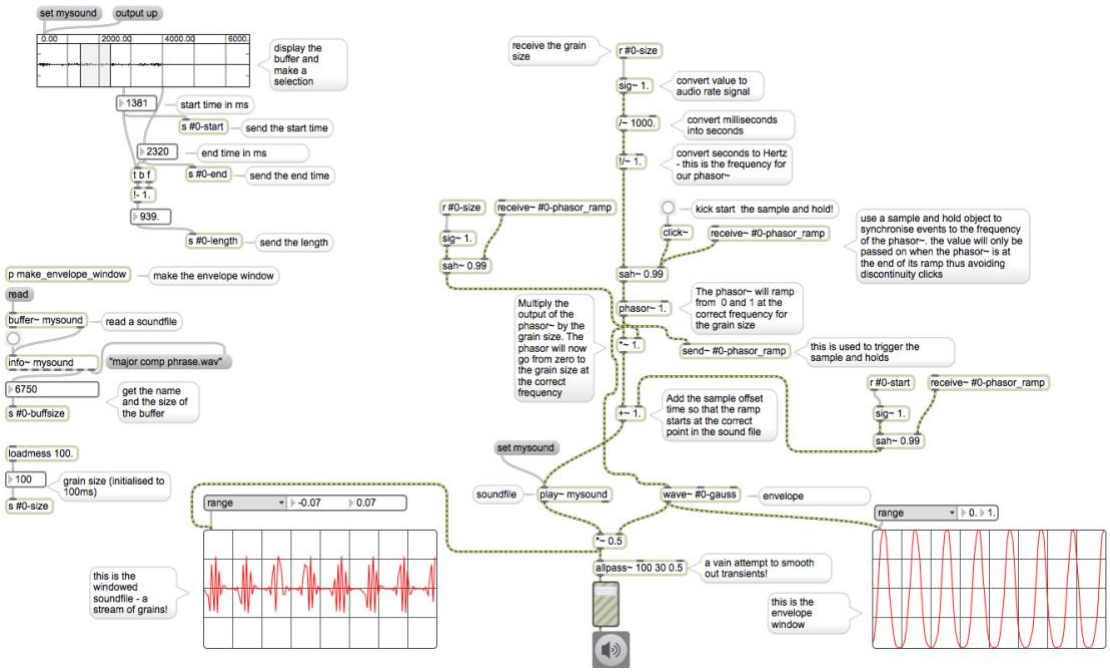

Figure 6: Granular synthesis patch adapted for sample accurate grain triggering

The patch in Figure 6 outputs a sample accurate stream of single grains. The position of the grains in the sound file is determined by the sample-offset value. At this stage the output of the patch is a rather harsh sound and further features need to be added to make it more musical before it can be used as the engine for a polyphonic granular synthesizer. There are also amplitude modulation artifacts in the sound at very small grain sizes and these need to be removed or at least masked. Having overlapping windows is one way of masking unwanted transients and artifacts. This is easily achieved by having a second play~ and wave~ pair driven by the same phasor~ but with its output ramp offset by 50%. Figure 7

[2] The timing stability of an audio rate system will still be subject to various factors both inside and outside the Max environment. These include the accuracy of the host computer clock, the signal and i/o vector sizes and the sampling rate.

(adapted from (McCloskey, 2011)) shows how this can be implemented. This could also be implemented using Max's pong~ object, which wraps a signal falling outside a specified range back into that range. Figure 8 shows how the phasor~ could be offset by 50% using the pong~ object.

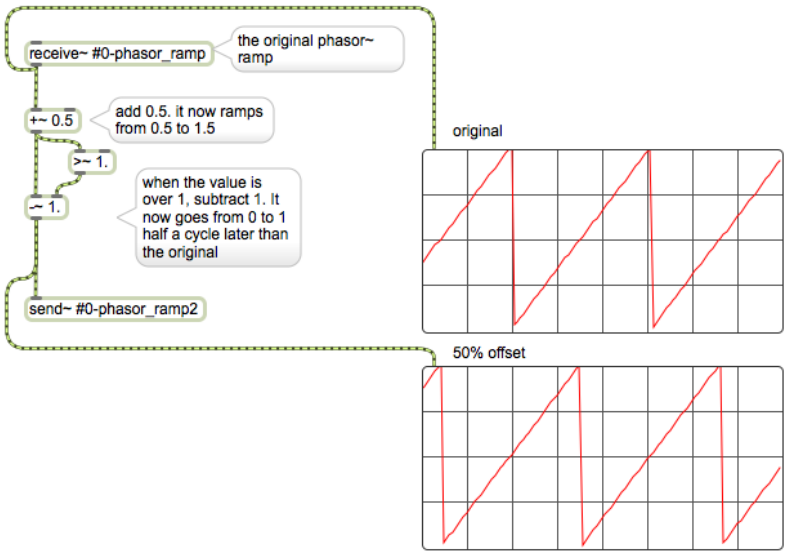

Figure 7: offsetting the phasor~ ramp by 50%

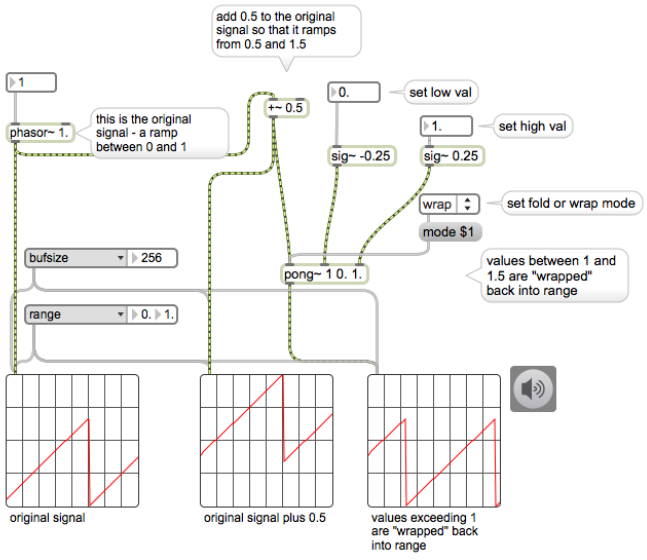

Figure 8: offsetting the phasor~ by 50% using the pong~ object

Adding a second, overlapped window significantly improves the sound quality. Routing the two windowed grain generators through allpass~ filters and mixing the sounds with the unfiltered output also helps to remove harsh artifacts.

Another way to improve the sound quality is to add a random value to the sample offset time. Small values help to blur the effects of the transients while larger values introduce increasingly chaotic results with potentially interesting musical applications. The sample

offset time can also be used to scroll through the sample. The start time, end time and length of a selection soundfile can easily be calculated and these can be used as messages to a line~ object which will output a ramp from the start time to the end time in the time of the selection. We can use the fact that the length of the selection is inversely proportional to the playback speed to manipulate the time taken to read through the sound file. Figure 9 shows a way of achieving this.

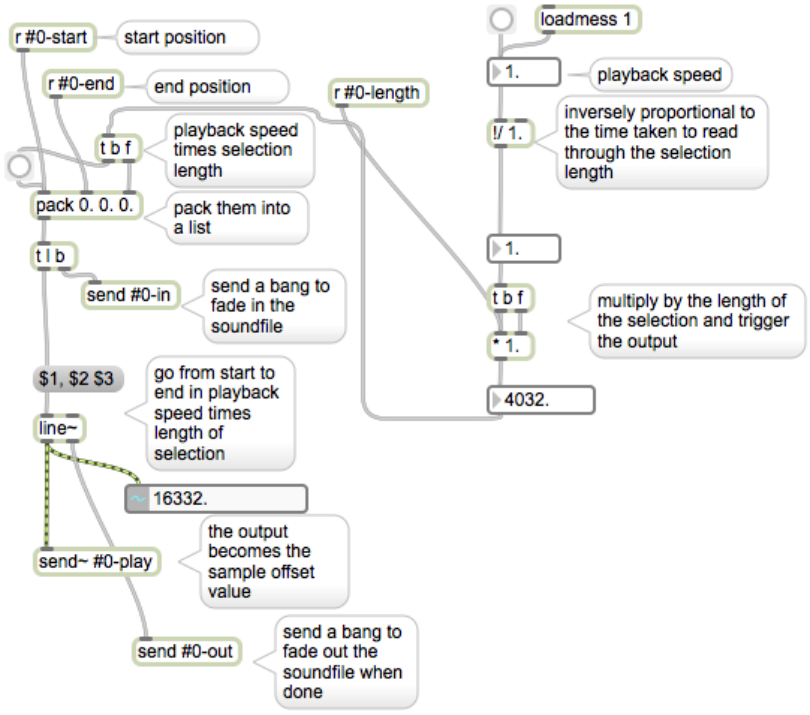

Figure 9: Controlling the playback speed to scroll through a selection of the soundfile

We now have a patch that will output a stream of overlapping, windowed grains controlled at the Max audio rate. We can read through a selection of a soundfile at a variable, user-defined rate or we can play a single grain repeatedly. There are controls to randomly offset the sample playback position and to change the grain size. We cannot yet, however, vary the density of grains that are playing. It would be useful to add controls to independently vary both the grain size and the interval between grains.

The zigzag~ object generates multi-segment linear ramps in a manner similar to the line~ object. It responds to input signals in different ways depending on the selected mode. In mode 1, an input signal can be used to trigger a ramp. If the input signal is set to be the grain density and the time taken to ramp from 0 to 1 is set to be the grain size, then zigzag~ can be used to read through the wavetable for the envelope window in the time of the grain size. The frequency with which this event is triggered corresponds to the grain density. Figure 10 shows how this is implemented.

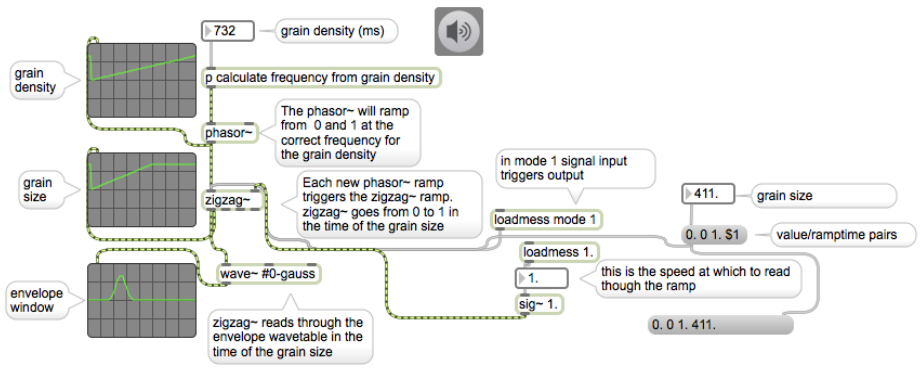

Figure 10: triggering the zigzag~ object with the phasor~ object to control grain size and grain density

Another way to achieve this in Max would be to use the train~ object. This object produces a series of pulses at a specified width and interval. The change~ object can then be used to report when the pulse signal is increasing or decreasing, i.e. when it is at the beginning or the end of a pulse. If the clip~ object is used to filter out the ends of the pulses, the pulses can be used to trigger a ramp in an object such as zigzag~ or Andrew Benson's shot~ external (Benson, 2013). This ramp will then read through the envelope window. Figure 11 shows how to implement this. An advantage of using the train~ object is that it is relatively easy to change the pulse interval and to interpolate between one interval and another without causing unwanted artifacts.

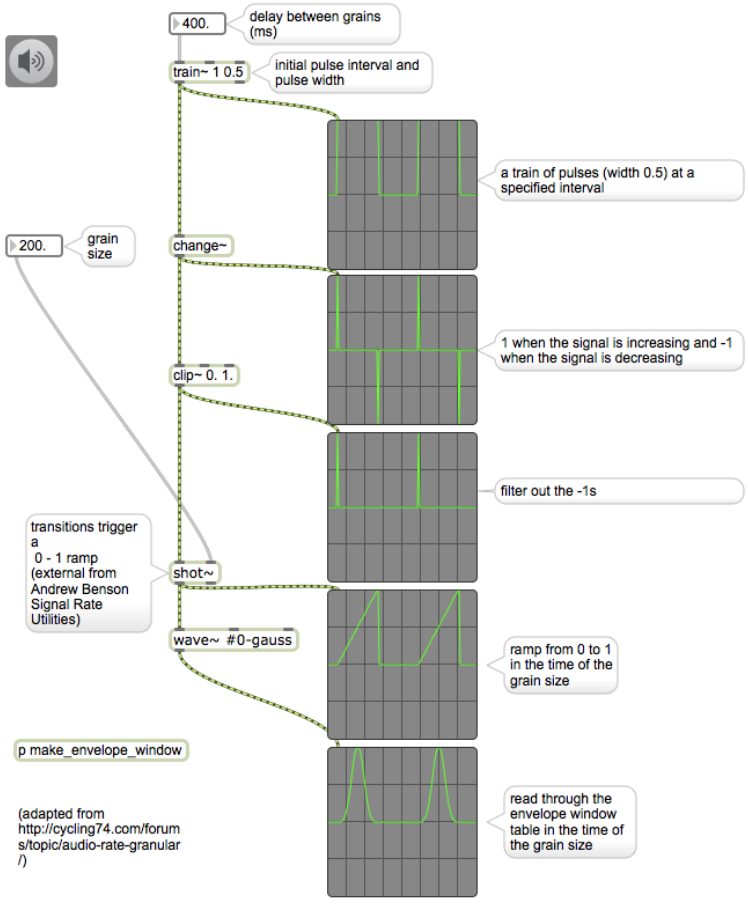

Figure 11: Using the train~ object to read through the envelope window

So far the only envelope we have been using has been the Gaussian window. There are, however, many other options and choosing a different envelope can have a significant effect on the tonal characteristics of the grain.

Hann window:

$$\omega(n) = 0.5\ (1 - \cos\ (\frac{2\pi n}{N-1})$$

Hamming window:

$$\omega(n) = \alpha - \beta(\cos\left(\frac{2\pi n}{N-1}\right))$$

Where α = 0.54 and β = 1 – α = 0.46

Blackman window:

$$\omega(n) = a_0 - a_1 \cos\left(\frac{2\pi n}{N-1}\right) + a_2 \cos\left(\frac{4\pi n}{N-1}\right)$$

(using a_0 = 0.42659, a_1 = 0.49656, and a_2 = 0.076849)

Blackman Harris window:

$$\omega(n) = a_0 - a_1 \cos\left(\frac{2\pi n}{N-1}\right) + a_2 \cos\left(\frac{4\pi n}{N-1}\right) - a_3 \cos\left(\frac{6\pi n}{N-1}\right)$$

Using a_0 = 0.35875, a_1 = 0.48829, a_2 = 0.14128and a_3 = 0.01168

Cosine window:

$$\omega(n) = \cos\ (\frac{\pi n}{N-1} - \frac{\pi}{2})$$

Using n as the sample index number and N as the window size, these windows can easily be created in Max using the expr object by applying the same technique describe in Figure 1 to create the Gaussian window. The patch can then be configured so that the user can easily select an appropriate window from a drop down menu and audition the results. A plot~ object can be used to display the window as shown in

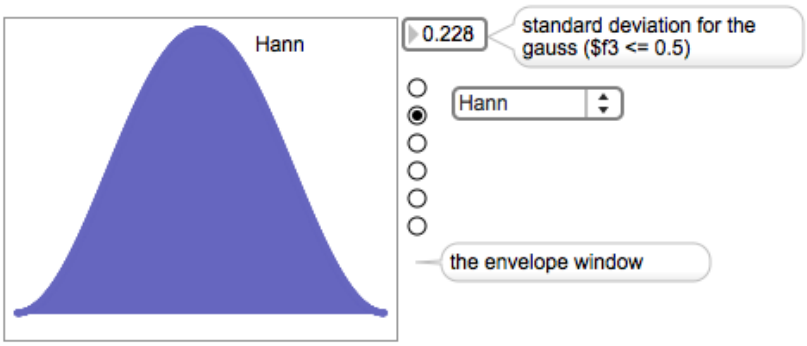

Figure 12: Using plot~ to display the envelope window

The audio rate granular synthesizer can be made polyphonic in the same way as the patch shown in Figure 4 and Figure 5. As before using the args argument in the poly~ object and prepending messages with #0 or #1 keeps messages local or global as appropriate. In order to avoid phase problems, the voice number for each instance of sub-patch loaded into ploy~ can be used to offset the phase of the phasor~ objects. For the polyphonic version of the sub-patch, the overlapping grain can be removed as this removes some unwanted artifacts. Spreading the grains across the stereo field also removes some unwanted artifacts. Inserting an equal power panning sub-patch at the end of the signal chain and sending a unique random pan position to each instance of the sub-patch is an effective way of creating a stereo image for each voice. The stereo panning sub-patch is shown in Figure 13. The patch uses a square root curve and has a dynamic setting, where the grain sweeps across the stereo image at a randomly generated frequency, or a static setting.

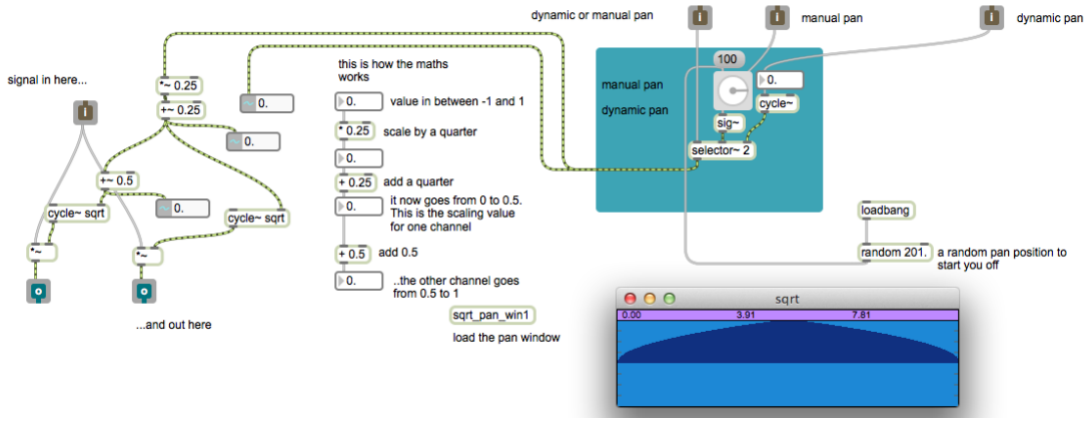

Figure 13: An equal power panning patch

A simple user interface for the granular synthesizer is shown in Figure 14. There are still many features that will be added to this. These will be described in later sections but the patch as it stands is a functioning and effective granular synthesizer.

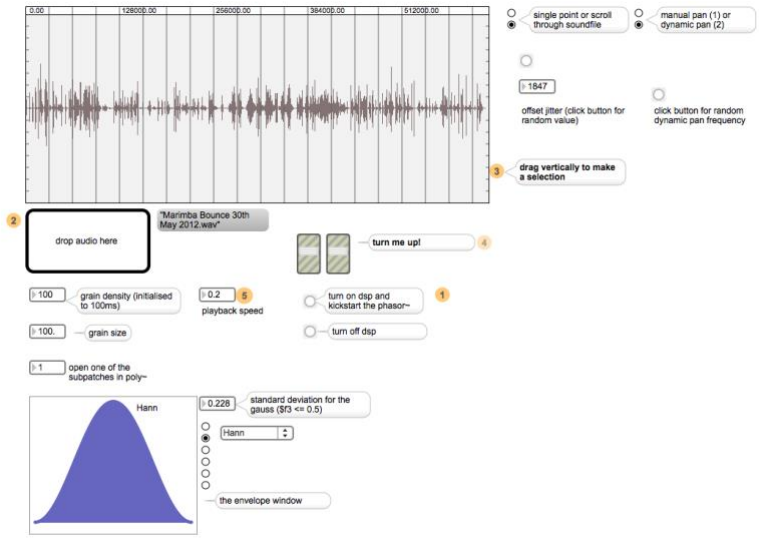

Figure 14: A functioning, audio rate granular synthesiser