Programming New Realtime DSP Possibilities with MSP

Christopher Dobrian

Gassmann Electronic Music Studio
School of the Arts — Music
University of California
Irvine, CA 92697-2775 USA
dobrian@uci.edu

ABSTRACT

The new MSP extension to the Max programming environment provides an easily comprehensible and versatile way to program realtime DSP applications. Because of its full integration into Max, MSP allows one to combine MIDI data and audio data readily in any program, and to hear the results immediately. This makes it an excellent environment for experimenting with new DSP algorithms and for designing music performances with a realtime DSP component.

This paper presents some algorithms for time-domain audio processing in MSP which are not commonly found in the repertoire of included effects for commercially available audio processors. These algorithms—which use the realtime segmentation of captured audio—are computationally inexpensive, yet are capable of producing a variety of interesting sonic effects. They include simulated time-compression and pitch-shifting of audio samples, segmentation of audio samples for use as “notes” in another rhythmic structure, and modulation to extreme rates of sample playback.

1. INTRODUCTION

MSP—written by David Zicarelli based on ideas of Miller Puckette—is the addition of audio signal processing capability to the existing *Max* programming environment. It provides an intuitive and versatile way to program realtime DSP applications, and has already become the chosen environment for such work among musicians. MSP presents at least two artistic advantages for a musician: it allows one to design and use unconventional DSP algorithms not readily available from commercial audio effects processors, and it allows a single program to produce many different musical results, dependent on the nature of the input or on decisions made in real time by the computer or by a performer. This is particularly appropriate for artistic works such as an audio installation located in a public space, or a musical performance that includes spontaneous improvisation.

In these pages I will explain selected algorithms for modification of digital audio in which the only method of processing is simply the unconventional playback of recorded sounds. The fact that these operations use stored audio does not necessarily mean that the processing is not effectively realtime. Since MSP can be programmed to automatically record incoming sounds, and begin playback and processing only milliseconds later (for all practical purposes at the same time as the sound is being recorded), these algorithms can be used on sounds that are performed live, and the control of the processing parameters can also be done in real time.

2. GRANULAR PLAYBACK OF RECORDED AUDIO

MSP allows one to capture incoming audio and store it either to disk or in RAM. As soon as it is stored in RAM it is available for access by any other part of the program, via a variety of playback methods. The algorithms explained below focus on three primary playback ideas: *1)* rapid access of very short segments of recorded audio (“grains” potentially as short as 1.5 milliseconds, but more commonly in the range of 20-100 milliseconds), *2)* segmentation of recorded sound into “notes” (usually longer than “grains”) which can be played rhythmically by Max, and *3)* use of a recorded sound as a wavetable for a lookup oscillator, such that the sound can modulate continuously from its original form into a periodic tone or vice versa.

2.1. Emulated phase vocoding in the time domain

The use of the Fourier transform for frequency-domain processes such as time compression/expansion and pitch-shifting is well documented. However, the fact that MSP runs in real time on a general-purpose computer (Macintosh PowerPC) means that frequency-domain operations involving Fourier transforms often tax the computer’s processing power significantly (with currently available processors), limiting the number of such processes one can use simultaneously. For this reason, in my own works that use MSP I have pursued less computationally expensive processing methods. One such method is the use of granular sample playback for simulation of time compression/expansion.

The conceptual basis of “granulation” is windowing small segments of an audio signal in rapid succession (often with some overlap of windows). In the implementation shown here (Figure 1), two overlapping repeating triangular windows are used. Identical triangular windows with a time offset equal to 1/2 the window duration are used in this case to maintain unity gain. In effect, a repeating triangular window is the same as amplitude modulation (multiplication) of the sound by a triangle wave with a DC offset (occupying the range 0 to 1). When two versions of this are added together, with the two triangle waves always 180° out of phase, the sum of the two triangle waves is always 1, so the effect of the amplitude modulation is nullified.

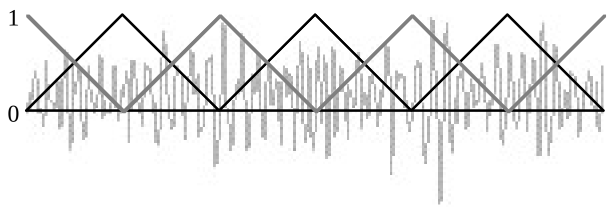

Figure 1. *Overlapping triangular windows on a sound*

The value of this windowing process is that each individual window (i.e., each cycle of each triangle wave) can be treated as an independent grain of sound, with its own unique playback speed and its own unique starting point within the original sound. For example, if each window is 4096 samples in duration, unmodified playback can be achieved by having each grain begin reading 2048 samples later in the sound than the previous (overlapping) grain. However, if each grain begins reading from the source sound only 1024 samples later than the previous grain, the entire sound will be traversed exactly half as fast the original. Of course the sound will be modified because, during the overlap, one grain will be playing a delayed version of what the other has just played. This results in comb filtering effects which can range from very subtle to very prominent, depending on the source sound and the delay between grains. The advantage is that by controlling only one parameter—how far the starting point of each successive grain leaps ahead in the source sound—the sound can seem to be compressed or expanded in time, and a variety of other effects such as echoes and comb filtering can be achieved.

This starting point incrementing parameter can be expressed as a multiplier of the normal (unmodified) leap size, which is 1/2 the window period. For example, a multiplier of 2 will cause the starting point of each grain to leap ahead in the source sound twice as far as normal, thus traversing the sound twice as fast, “compressing” it by a factor of 2. A multiplier of 0.5 will make the leap half as large as normal, causing the grains to traverse the source sound at half the original tempo.

Furthermore, each grain can itself be played at any increment rate as it reads the source sound, thus changing the internal speed of each grain and transposing its pitch. If, for example, the grains are played with a *transposition* value of –12 semitones (i.e., at half speed) with a *tempo* factor of 1, the effects is similar to pitch shifting down one octave. The sound of each grain is slowed down, but the grains progress through the source sound at a normal rate. The trade-off in quality is that some parts of the source sound are left unread (in the case of downward transposition without a corresponding change in the tempo factor) or overlapped (in the case of upward transposition). Again, depending on the nature of the source sound and on the amount of the transposition, the effect may be either subtle or extreme.

This particular implementation of granulation (chosen from among many different possible approaches) has two significant advantages: *1)* the input parameters for varying the process have a direct relationship to musical attributes—transposition and tempo—which correspond to the sonic effects one expects from pitch-shifting and time compression/expansion, and *2)* one can modulate from a completely unmodified playback of the original sound (transposition=0, tempo=1) to a wide variety of modifications and effects. To do this, we need a continuous chain of triangular windows, overlapping precisely as shown in Figure 1, and we need to be able to make changes in the starting point and speed of each grain at precisely the moment when the amplitude of the window is at 0 (in order to avoid clicks caused by discontinuities in the output sound).

However, achieving the sample-accurate control necessary to realize this idea correctly is not obvious in MSP. MSP calculates a *vector* of several milliseconds worth of samples at one time, and control information from Max—such as unique starting point and transposition values for each grain—can only be supplied at the beginning of each vector calculation. For this reason, I have chosen to express the offset between grains as an integer multiple of the signal vector size, and the length of each grain (each triangular window) is twice that. By looking for the end of the window (testing for the maximum sample value coming from a **count~** object), the **edge~** object sends out a bang which can be used to trigger new control values at the beginning of the next vector (with the *Scheduler in Audio Interrupt* option checked to ensure that the control information is always synchronized with the beginning of a new vector).

The implementation shown in Figure 2 is a bit complicated to read without explanation, so I will point out its primary features. Because it is designed to be used as a subpatch in a larger program, certain precautions have been taken which make it easy to re-use in multiple contexts and/or multiple copies. For one thing, the buffer that contains the source sound is not included in the subpatch; the **wave~** objects refer to a **buffer~** that can reside in any loaded program, the name of which can be specified as an argument or sent in the second inlet. Similarly, the triangular windows are not read from a **buffer~**, but are instead calculated by a mathematical formula (in the bottom part of the example); this prevents confusion that could possibly arise from creating multiple instances of a **buffer~** with the same name. As an additional precaution, this example eschews the use of a **delay~** object, even though that would be the easiest way to make the precise sample offset for the two overlapping triangular windows. Use of **delay~** for this purpose would limit the maximum window size (since the amount of RAM set aside for the **delay~** must be specified as an initializing argument), and that memory could also add up quickly if multiple copies of this subpatch are used. Avoiding **delay~** requires some machinations to keep both windows synchronized; they are derived from the same looping sample counter (the **count~** object), but the offset of one window must be recalculated for every 1/2 window length. The initial starting point in the source sound is specified as an integer number of samples received in the left inlet. This triggers a calculation of the window size based on the signal vector length,
and starts the sample counter looping from 0 to *windowsize-1*. To determine the speed of each grain, this sample count is multiplied by a factor derived from the current *transposition*

value, and the starting point offset is added to that. At the end of each half-window (detected by the ==~ and **edge~** objects), each successive necessary starting point is calculated (and is multiplied by the *tempo* factor).

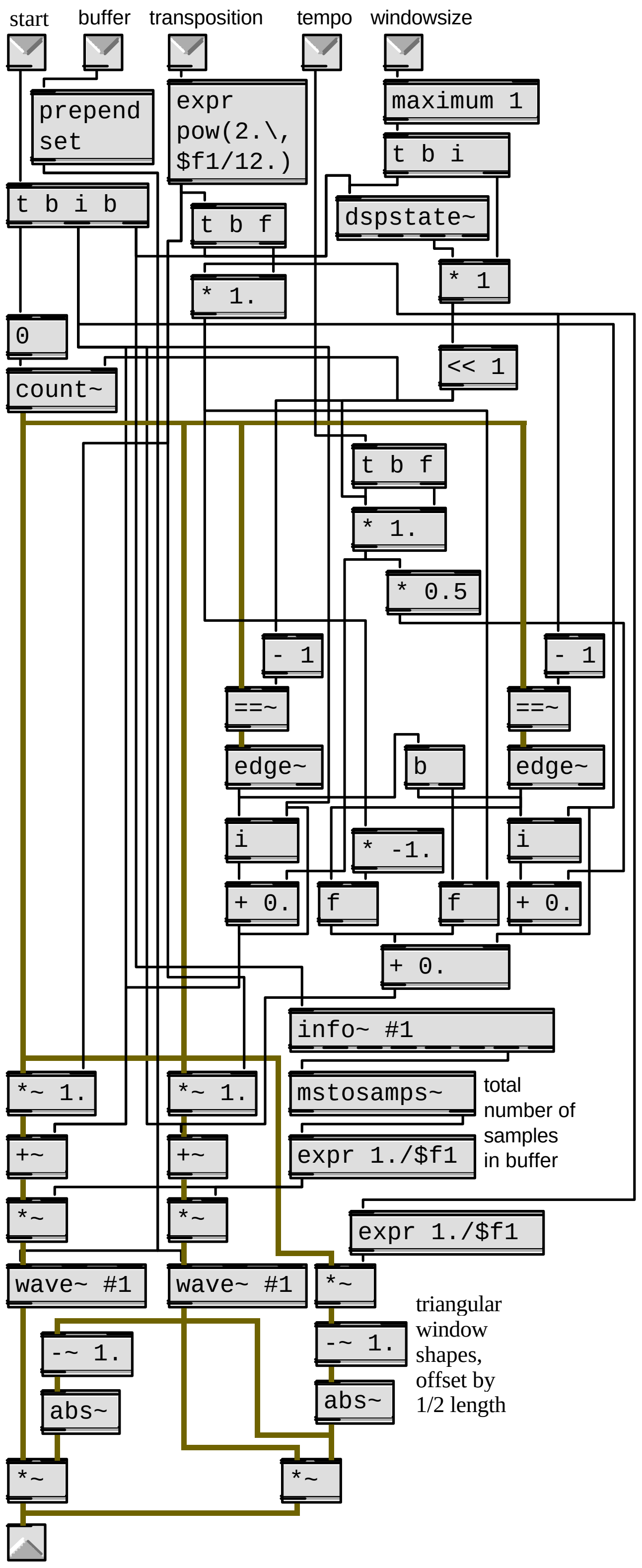

Figure 2. *Control of grain tempo and transposition*

2.2. Rhythmic segmentation of recorded audio

Instead of using tiny grains of sound (as in most types of granular synthesis), one can divide a sound into slightly longer segments more on the order of short notes. These notes will have the timbral characteristic of some small portion of the source sound, but can be used in any desired musical structure. By making the minimum note length equal to the fastest pulse in an arbitrary rhythmic structure, one can impose any desired rhythm on a recorded sound. Figure 3 shows a way to do this with ordered segments of a sound, leaving the source sound essentially intact and recognizable.

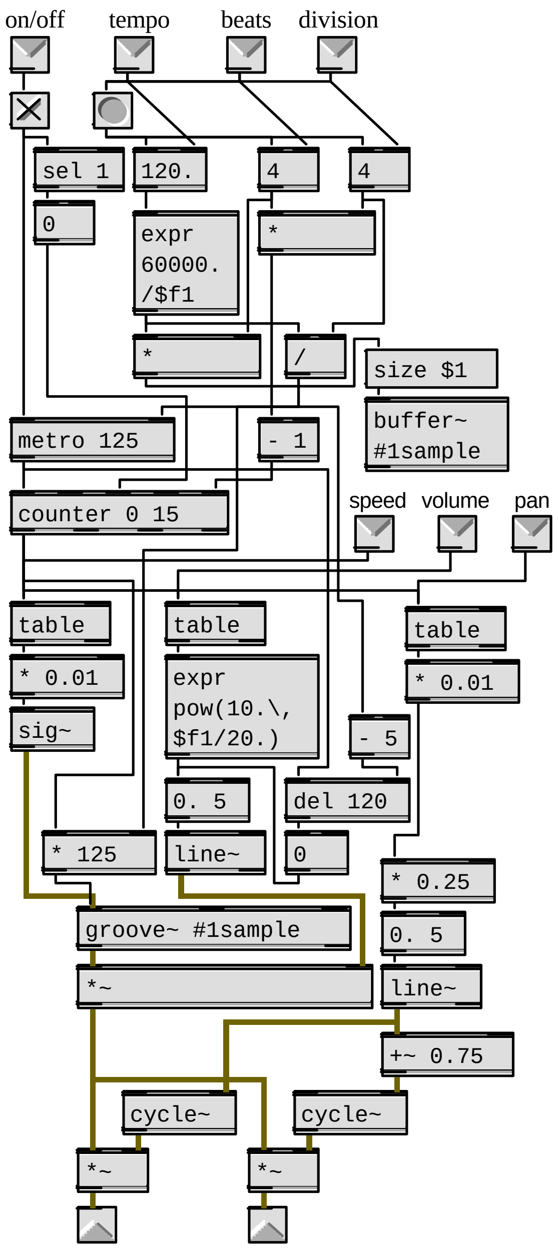

Figure 3. *Sample played with rhythmic segmentation*

The inlets allow one to specify a tempo and a metric structure—beats per measure and divisions per beat—and these values are used to calculate the size of the buffer that is needed, the duration of each segment (beat division) and the speed of the

metro object that triggers each segment. (It is assumed that the contents of the **buffer~** are set by a **record~** object elsewhere in the program.) This example has initial default values of 120MM, 4 beats, divided in 4 parts per beat. For each segment of the sound (each beat division), a **counter** reads from a **table** to get a stereo panning value (used to calculate the level for each of two **outlet**s), an amplitude (specified in dB), and a playback speed. These values can be supplied to the **table**s with *set* or *refer* messages. Thus, the tables can at any moment be filled with an entirely new rhythm (as delineated by pitch, loudness, and stereo location) in any desired tempo and meter. In this way, any source sound can be used, yet an arbitrary rhythm can be imposed upon it to achieve interesting musical effects.

2.3. Sample as waveform

The length of a sound that can be stored in RAM by MSP (in a **buffer~** object) is limited only by available application memory. The most common uses for a **buffer~** are *a)* storage of a very small segment of audio (e.g., 512 samples) for use as a lookup table by a periodic oscillator (a **cycle~** object), and *b)* storage of a longer segment (either pre-recorded or recorded in real time) for "sampling" or other less conventional playback methods such as those shown here. However, in MSP it is a simple matter to use a buffer of any length for either purpose. For example, by attaching a **phasor~** object to a **wave~** object, or by attaching a scaled **phasor~** object to a **play~** object, one can traverse an entire buffer periodically at any rate (Figure 4).

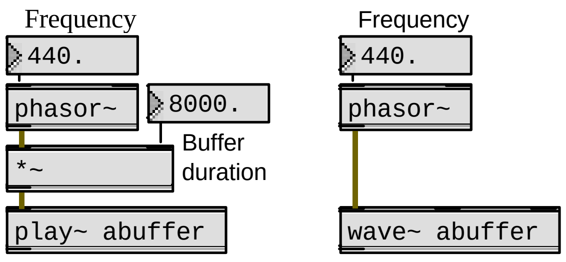

Figure 4. *Two ways to use entire buffer as a wavetable*

Although the frequency of the **phasor~** is known, the actual frequency content of the output depends on the contents of the buffer, and can be difficult to predict when a long and complex buffer is used. Since the frequency of the **phasor~** can be varied continuously between audio and sub-audio rate, a continuous transformation can be made from periodic tone to unaltered playback of the buffered sound at its original rate. Figure 5 demonstrates one example of such a transformation. It is an automated process that plays repeated "notes" every 125ms, using a **phasor~** and **wave~** combination that reads through a 2-second buffer. With each successive note, the frequency of the **phasor~** is reduced, beginning at 64Hz and ending at 0.5Hz (the appropriate rate to play a 2-second buffer at its original speed). Every 16 seconds, a new buffer is selected and a new 14-second downward glissando is begun; at the end of the glissando, the buffer is played in its entirety for a full two seconds.

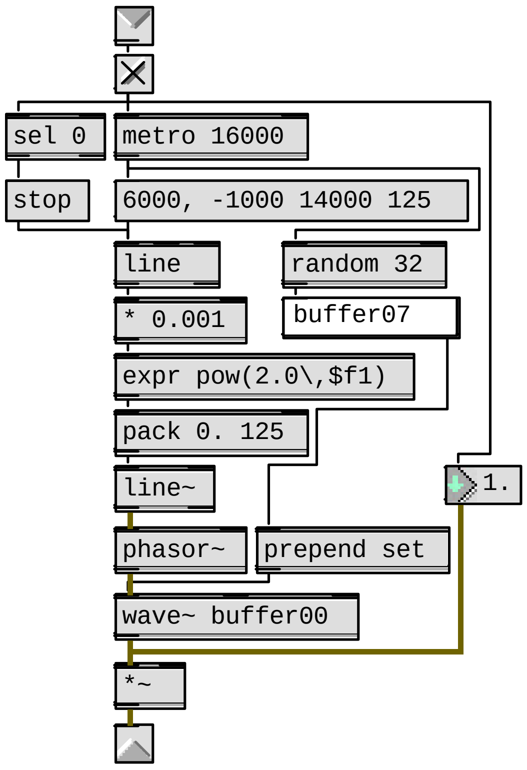

Figure 5. *Transition from wavetable to sample*

The above example uses a pre-established transition from periodic tone to unaltered playback. Such modulation could equally well be supplied in real time using input from a MIDI controller or any other Max control data.

3. CONCLUSION

I have demonstrated here three computationally inexpensive methods of processing pre-recorded sound (or sound captured only a few milliseconds earlier) which are not commonly used in commercial effects-processing systems. They employ the segmentation of sound for granular synthesis, simulated time compression/expansion, simulated pitch shifting, wavetable lookup, and rhythmic performance of contiguous segments of a sound. These processes use modification parameters specified in "musical" terminology (*transposition*, *tempo*, *beats*, etc.) making them easy to incorporate in an algorithmic performance or audio installation.

4. BIBLIOGRAPHY

Dobrian C., *MSP: The Documentation*, version 1.1, Cycling '74, Santa Cruz, California, 1998.

Dolson, M.,.

Roads, C., *The Computer Music Tutorial*, MIT Press, Cambridge, Massachusetts1996.