Sight Unseen

A dissertation submitted to the

Graduate School

of the University of Cincinnati

in partial fulfillment of the requirements for the degree of

Doctor of Musical Arts

In the division of Composition, Musicology, and Theory

of the College-Conservatory of Music

March 28, 2017

By

Michael Lukaszuk

Master of Music, The University of Western Ontario, 2013

Bachelor of Music in Theory and Composition,

The University of Western Ontario, 2011

Committee Chair: Dr. Mara Helmuth

Abstract

This document discusses compositional ideas and computer music techniques employed by the composer, resulting in the stereo fixed-format electronic work *Sight Unseen*. The design and use of new tools for working with granular synthesis should be of particular interest. The use of the typewriter recordings, and the human voice, notably the way in which speech is obscured via computer manipulation is also another important aspect of the composition discussed in this document. The treatment of voice recordings relates to the theme of information literacy, which is explored in the music with some depth.

Acknowledgements

I would like to thank my advisor Dr. Mara Helmuth for her tremendous guidance throughout my time at CCM. I would also like to thank my wonderfully understanding girlfriend Haerim for all of her patience throughout the arduous journey that culminated in this piece.

Table of Contents

I. Program notes

Sight Unseen can be thought of as a series of songs and interludes that amalgamate into a single work. There are sections that prominently feature processed recordings of voices and a typewriter (an instrument also used to transmit text), and others that can be thought of as abstract musical reactions to those previously mentioned sections. Although the programmatic element is not the only important aspect of the piece, the theme of information literacy is explored with some depth. Information literacy refers to the ability to evaluate the quality of information and understand when information is needed and how it can be located. This might seem like a rather dry topic but it is tremendously relatable to various aspects of our everyday lives. Having grown up with print books and online resources like the Encyclopedia Britannica, I really did assume that what I read or heard was surely trustworthy. Now it seems as if I have to really understand the larger context in which information is disseminated because sources are so often obscured. While certain sections deal with the extra-musical theme quite directly, overall, the piece is about conveying an emotional reaction – to express the frustration, confusion and other feelings that come with not being able to trust or understand the validity or context of information.

II. Computer Music Techniques

Granular Synthesis

The term Granular Synthesis refers to a technique used heavily in computer music composition where high densities of small acoustic events called

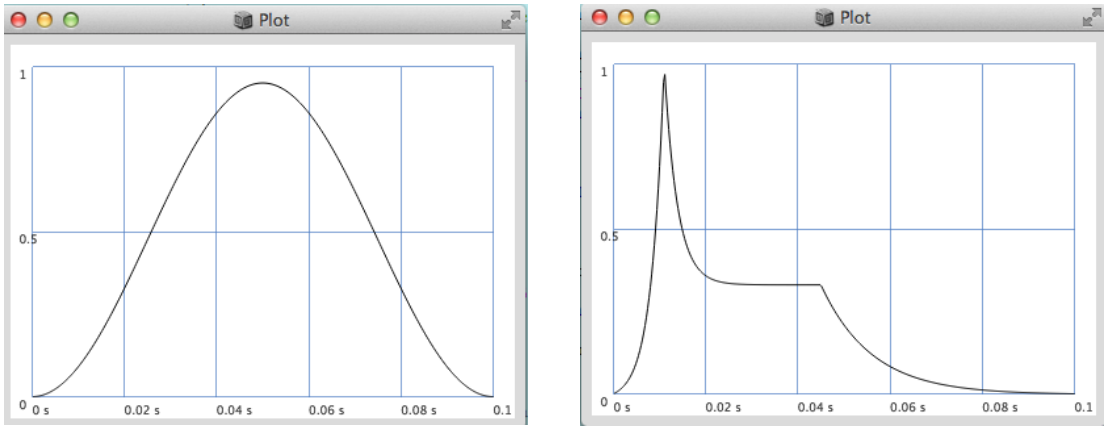
grains are accumulated to form larger textures. Typically grains last from 10-100 milliseconds. They are microsonic acoustic particles, meaning that the human ear cannot distinguish the specific duration, timbre and other characteristics of the grain. The overall sonic character, especially the timbre of the resultant granular texture is the result of manipulations on a variety of parameters belonging to individual grains or groups of them, including duration, frequency, grain phase, inter-grain delay (the amount of time before the grain is heard after the previous one), and the amplitude envelope that is typically imposed on the grain, often referred to as the grain window. As granular synthesis pioneer Barry Traux mentions on his website, "What is most remarkable about the technique is the relation between the triviality of the grain (heard alone it is the merest click or 'point' of sound) and the richness of the layered granular texture that results from their superimposition."[1] In other words, the manipulations of parameters of single grains have large implications for the overall sound we hear when using this technique.

The amplitude envelope imposed on the grain is often used to avoid a click at the beginning and end of the grain, ensuring a smooth fade in and fade out of the particle. Much of my music that employs granular synthesis techniques explores the effect of manipulating the slope of the envelope, or its overall shape to drastically change frequency spectrum of individual grains or larger collections. In imposing an amplitude envelope on the grain you are also limiting the bandwidth of its individual frequency spectrum. The grain envelope exemplifies

[1] Truax, Barry, "Granular Synthesis." https://www.sfu.ca/~truax/gran.html (accessed February 12, 2017)

how granular synthesis blurs the distinction between the time and frequency domains. Smooth curves give a well-rounded effect, shapes with jagged corners act like a series of envelopes imposed on a single grain giving rich high-frequency content.

Fig. 1 a-b, smooth curve (similar to hanning envelope) vs. atypical shape with jagged edges

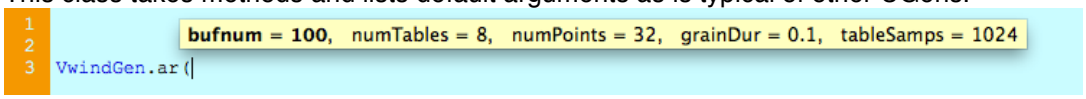


In composing this work I developed a few unique tools that deal with the creation or use of grain envelopes in ways that, at least in my opinion, have yet to be explored. These include two pseudo UGens for the SuperCollider audio programming language, GrainsMPL and VwindGen, and an instrument for RTcmix, a scriptable software for sound synthesis and processing.

The main building block for developing tools in SuperCollider is the UGen. It is comparable to an instrument in RTcmix, and to a lesser extent, an object in Max/MSP. UGens are classes that calculate signals and dictate server behavior. In SuperCollider, pseudo UGens are classes that inherit from existing classes. They allow the user to greatly extend existing class(es) to provide new features and functionality. With this approach you are creating your own UGen that functions just the same as any other one, you specify methods and arguments

and add help files. The advantage with this approach is you can create powerful tools without having to delve into C++ plugin code. The disadvantage is that you cannot produce any new types of audio processing or server behavior, confined to reworking existing classes and functions already belonging to the language.

Fig. 2 shows a pseudo UGen – the language interpreter responds to it just like any other class. This class takes methods and lists default arguments as is typical of other UGens.



GrainsMPL inherits the GrainSin class. GrainSin is a relatively straightforward UGen that performs granular synthesis using a sine tone as a synthesis waveform. It deals with standard grain parameters such as trigger rate, duration, etc. Its envbufnum argument is the reason behind choosing this class as the model for my own UGen. This argument asks for a signal inside of a buffer that is to be used as the grain envelope/window. Providing the number -1 generates a hanning window – the hanning window is derived from taking a cosine shape and raising a portion of the curve so that everything is above zero on the y-axis. This is one of the most common envelope shapes used in granular synthesis. GrainsMPL adds a great deal of new functionality to this argument and adds a variety of entirely new arguments to create a considerably more powerful tool than GrainSin for exploring windowing (grain envelopes) in granular synthesis.

The lines of code (fig. 3, lines 2-3) following the “arg” keyword in dark blue show the arguments that GrainsMPL takes along with default values. Of particular interest should be the final four: windType, windPoints, windStart and

windGrow. Please see fig. 4 for a basic summary of all of the arguments that GrainsMPL takes.

Fig. 3, code for GrainsMPL pseudo-UGen. Once your class library has been recompiled you can use the instrument as any other UGen.

```
GrainsMPL {
    *ar {arg  grainLayers = 1,trig = 10, trigTight = 0.2,dur = 0.1, freq = 330, pan = 0,
         maxGrains = 1024,mul = 0.2,windType = 0,windPoints = 4,windStart = 0.015, windGrow = 0.01;

        var out, winenv,winenv2,winenv3,winenv4,winenv5,z,s, zChoose,zEnv,windTimes,windArr,windTypesel;

        s = Server.default;

        windTimes = dur/windPoints;

        winenv = Env(Array.fill(windPoints,{rand(1.0)}),Array.fill(windPoints-1,{windTimes*rand(0.25)}),Array.fill(2,{(rand(10)-5)}));

        winenv2 = Env(Array.geom(windPoints,windStart,windGrow),Array.fill(windPoints-1,{windTimes*rand(0.25)}),Array.fill(2,{(rand(10)-5)}));

        winenv3 = Env(Array.series(windPoints,windStart,windGrow),Array.fill(windPoints-1,{windTimes*rand(0.25)}),Array.fill(2,{(rand(10)-5)}));

        winenv4 = Env.triangle(dur,0.95);
        winenv5 = Env.perc(dur/10,dur-(dur/10),curve:rand(8)-4);
        //uses the .perc method of the Env class to window grains according to an Attack time-> release time, sustain level, slope

        windTypesel = windType;


        windArr = [0,1,2,3,4]; // use the windType argument to select from different grain windows via this array
        zEnv = switch(
        windArr.indexOf(windTypesel),

            0, {zChoose = winenv},
            1, {zChoose = winenv2},
            2, {zChoose = winenv3},
            3, {zChoose = winenv4},
            4, {zChoose = winenv5});

        z = Buffer.sendCollection(s, zChoose.discretize, 1);// zChoose is the resultant grain window

        out = Mix.fill(grainLayers,{GrainSin.ar(1,GaussTrig.kr(trig,trigTight),dur,freq,pan,z,maxGrains:maxGrains,mul:mul)});
        ^out;
    }
}

```

Fig. 4, help file for GrainsMPL pseudo-UGen

GrainsMPL
granular synthesis using sine tones with a variety of controls for grain window generation

Class methods: *ar

Arguments:

grain layers	defaults to 1. You can multiple layers if you're incorporating randomness in other areas.
trig	how many grains per second. Doesn't require a control signal to trigger, just give an int.
trigTight	amount of randomness in the trigger rate. 0.0-1.0, 1.0 = maximum possible randomness.
dur	grain duration, not duration of overall stream/cloud. Defaults to 0.1 per. second.
freq	frequency of grains in Hz.
pan	grain pan.
maxGrains	maximum number of overlapping grains that can be used at a given time.
mul	amplitude multiplier. Defaults to 0.2.
windType	refers to the specific algorithm used to generated the grain window. Use int. 0-4.
windPoints	number of breakpoints in the wavetable that is created via windType.
windStart	if windType is 1 or 2, provides the starting value for the wavetable.
windGrow	if windType is 1 or 2, provides the value by which each successive point will grow.

windType input:
0 = randomly generated points and times that can be controlled via the windPoints argument
1 = geometric series that grows according to windGrow and will continue according to limit expressed in windPoints
2 = same as 1 but using arithmetic series.
3 = uses triangular grain env
4 = uses a grain envelope modelled on the attack and decay of a percussion instrument: attack, release and slope of the gradual release

Example:

```
SynthDef("MPLgrainTest", {

        var out1,out2,trigRate,grainDur,grainFreq,pan,windType;

        trigRate = TRand.kr(2,12,Impulse.kr(5));
        grainDur = 0.1;
        grainFreq = ([770,380]);
        pan = TRand.kr(-1,1,Impulse.kr(grainDur));
        windType = 3;

        out1= GrainsMPL.ar(1,trigRate,0.2,grainDur,grainFreq,pan,1024,0.5,windType,6,0.025,0.05);
        out2 = GrainsMPL.ar(3,trigRate*MouseX.kr(0.25,2.0),0.2,grainDur,
                cubed(reciprocal(grainFreq)*3000),pan,1024,0.5,windType-3+rand(4),8,0.015,0.05);

        Out.ar(0,[out1,out2])}).add;
```

VwindGen (variable window generator) is another pseudo-UGen that I designed to create many of the sounds used in the piece. Essentially, what it does is combine components from a series of wavetables, recording that signal into a buffer that holds the single aggregate envelope for a single grain. VwindGen controls the number of tables used and a couple features regarding the design of the aggregate wavetable such as the rate at which the instrument interpolates across a series of wavetables. Once this UGen is evaluated using a SynthDef, the server will recognize the aggregate waveform according to the ID number provided in the VwindGen bufnum argument. Please see fig. 5 for more information.

In a sense this approach can be thought of as a variant on the kind of distortion resulting from waveshaping synthesis, but transforming a grain envelope instead of a synthesis waveform.

Fig. 5, code for VwindGen pseudo-UGen

```
VwindGen {
    *ar {

        arg bufnum = 100,numTables = 8,numPoints = 32, grainDur = 0.1,tableSamps = 1024;

        var s,numbTables,b,x,sampleTosec,durtoHz,recordVenv,envBufOut,a,bufoffset,tableDur;


        tableDur = tableSamps/48000; // unused at present

        s = Server.local;

        numTables.do
        ({  arg i;
            var a;
            s.sendMsg(\b_alloc, i, tableSamps);
            // at this time table size (in samps) has to be a power of 2

            // generate array of harmonic amplitudes
            a = Array.fill(numPoints,0);
            numTables.do({ arg i; a.put(numPoints.rand, 1.0) });
         // fill table
       s.performList(\sendMsg, \b_gen, i, \sinel, numTables, a);
       });


    b = Buffer.alloc(s,48000 * grainDur,1,bufnum:bufnum);

    bufoffset = 0;

    sampleTosec = 48000 * grainDur;

    durtoHz = reciprocal(grainDur);

    x = EnvGen.kr(Env.circle([0,numTables,1],[grainDur/2,grainDur/2],\lin),doneAction:2);
    recordVenv = VOsc.ar(bufoffset+x,durtoHz);
    envBufOut = RecordBuf.ar(recordVenv,bufnum,doneAction:2);
                ^envBufOut;
    }
}
```

Fig. 6, help file for VwindGen pseudo-UGen

VwindGen
generates a single wavetable for grain envelope/window that resulted from interpolation across a series of wavetables that the UGen generates.

Class methods: *ar

Arguments:

bufnum	an explicitly stated buffer ID. This number is input for env. in granular UGens
numTables	the number of tables that will be generated to sweep across to produce the final table
numPoints	the number of breakpoints in the table
grainDur	the duration (in seconds) of the final table
tableSamps	the duration (in samples) of the tables that are created via the numTables argument

Many granular synthesis UGens (e.g. GrainSin, GrainFM) have an envbufnum argument for where a signal should be placed that is to be used as the grain envelope. The corresponding bufnum from the use of VwindGen should be used as the envbufnum argument with such UGens. This instrument does not perform any sound synthesis, it is a tool for generating interesting grain windows.

Fig. 7, use of VwindGen in a SuperCollider SynthDef

```
SynthDef("Vwind1", {
    var out1;
    out1 = VwindGen.ar(101,3,28,0.05); // sweep from tables 1 -> 3 -> 1 over the course of 0.05
}).add;



Synth("Vwind1");

SynthDef("VwindGrains", {
    arg bufoffset = 0;
    var out1,out2;
    out1 = GrainSin.ar(1,Impulse.kr(5),0.05,330,0,101,512,0.2)!2;
    out2 = GrainFM.ar(1,Impulse.kr(8),0.05,110,80,15,0,101,mul:0.1);
    Out.ar(0,[out1,out2])}).play;

```

MIKEGRAN, the third tool heavily used for granular synthesis in my piece, is an RTcmix instrument that randomly chooses from a variety of synthesis waveforms and grain envelopes. These are provided as tables via makegen functions. It also has some interesting presets for quasi-synchronous grain rates. The C++ code for this instrument can be thought of modified and extended version of elements from the SGRANR and WAVETABLE instruments that come with RTcmix.

Fig. 8, shows how random grain envelopes are selected in the C++ code

```
mikeRand2 = new Orand;

float grainEnvtest = int (mikeRand2 -> range(1,5));

// These lines are for randomly choosing from the grain envelopes
// in makegen function slots 5-9. Uses Orand.

if ( grainEnvtest == 1)
    realgrainEnv = grenvtable;

else if (grainEnvtest == 2)
    realgrainEnv = grenvtableTwo;

else if (grainEnvtest == 3)
    realgrainEnv = grenvtableThree;

else if (grainEnvtest == 4)
    realgrainEnv = grenvtableFour;

else // so if mikeRand2 chooses 5
    realgrainEnv = grenvtableFive;
```

Fig. 9, demonstrates presets for grain rates

```
// variation on L system - cantor set -  if rateType = 4

float cantorA = durlo/30;

float cantorB = durlo/30;

cantorA = cantorA + cantorB + cantorA;

cantorB = cantorB + cantorB + cantorB;


// if rateType = 5 variation on fibonnaci sequence  algorithm

fibbonDur = durlo/50;

fibbonDur = abs((fibbonDur - 0.01) + (fibbonDur -0.02));

fibbonDur = fibbonDur + 0.001 ;


if (rateType == 0)
    realRate = rate;
else if (rateType == 1)
    realRate = rate * rrand();
else if (rateType == 2)
    realRate = rate+realRate/4;
else if (rateType == 3) //
realRate += algaeA;
else if (rateType == 4)
    realRate += cantorA;
else if (rateType == 5)
    realRate += fibbonDur;
```

The idea of quasi-synchronicity is at the heart of the MIKEGRAN instrument. There is a great deal of focus on extremely asynchronous grains in contemporary electronic and computer music, and there already exist many tools for synchronous granular synthesis. I wanted to have something that allowed for a tightly controlled and somewhat predictable grain trigger rate and selection of waveforms and grain envelopes.

Comb Filters

While many common filter types used in electronic and computer music are implemented using a cutoff or center frequency to adjust the bandwidth of frequencies amplified or attenuated, comb filters rely on feedback and minute

delay times to produce a series of peaks and troughs in the frequency spectrum of a sound.

In composing this piece I relied on two tools for comb filtering: 1) the GRM Tools Comb Filters plugin, 2) some simple comb filtering patches made in SuperCollider that loop audio files and apply the CombL UGen with small arrays of delay and decay times.

Compositionally the GRM Tools plugin was useful because it allows the user to specify the desired fundamental frequency for each comb filter. In many other comb filter tools, the user specifies a delay time that will affect the frequency spectrum. The fundamental frequency of the delay time in a comb filtering module is the inverse of the delay time, so it is not terribly difficult to understand what kind of spectral effects you will produce with a particular delay time, it is just simply much more convenient, especially when using multiple comb filters, to be able to input a fundamental frequency. In my piece, this allowed me to create kinds of triads and other such chords in the frequency response of the filter. Such sonorities were used more like “sound objects”, there is no functional tonality with regards to my use of “triads” in comb filter frequencies.

Using a variety of simple patches similar to the one below, I was able to add looping and phase-shifting effects while applying the comb filter to the contents of the buffer. Using different playback rates and triggering looping on and off, the comb filtered sonorities accumulated into a variety of interesting rhythmic patterns.

Fig. 10, tertian sonorities using the GRM Tools Comb Filter plugin

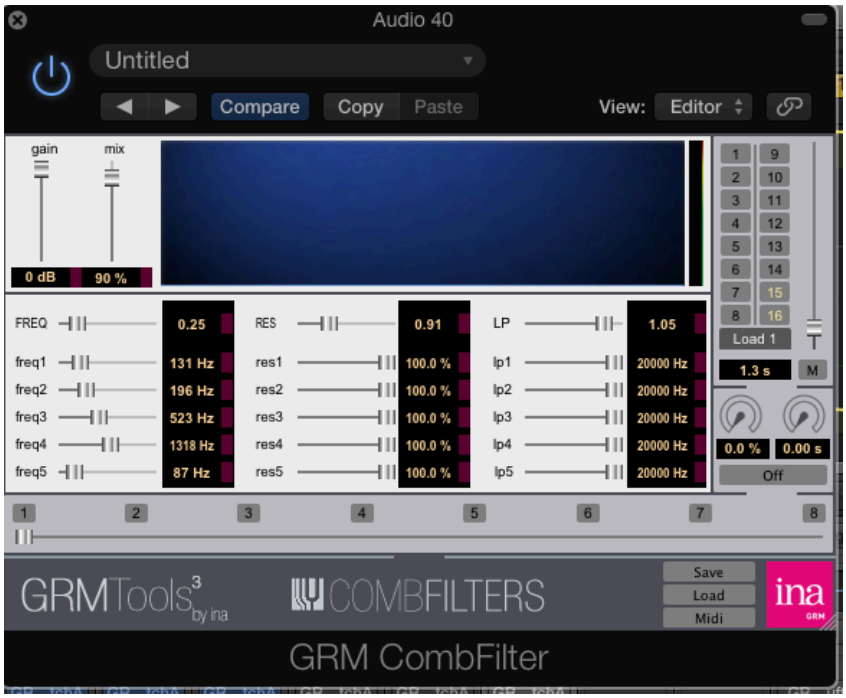


Fig. 11, comb filter patch with looping and LFO capabilities to allow for phase shifting-type effects

```
SynthDef("moreComb", {
    var out1,b,out2;

    b = Buffer.read(s, Platform.resourceDir +/+ "/sounds/Mikegran_cube_mono_9.aiff");
    out1 = CombL.ar(PlayBuf.ar(1,b,4.0,loop:1),0.1,[0.002,0.0004],[1.5,2.0],mul:0.0015*SinOsc.kr(0.1));
    out2 = CombL.ar(PlayBuf.ar(1,b,5.0,loop:1),0.1,[0.00066,0.00033],[1.5,2.0],mul:0.0006*SinOsc.kr(0.05,add:1));

    Out.ar(0,[out1+out2,out1+out2])}).play;
```

Treatment of Recorded Sound

Recorded sound, particularly the sound of my own voice and sounds from a typewriter play an important role in the piece. Compositionally I experimented with various forms of effects processing such as bit-depth reduction and various filters, and spatialization within the stereo field was also an important means of treating the recorded samples to reflect the creative outlook of the piece. The

basic idea is that the recordings are processed in a way that often obscures the text or and the perceived location from which the speaker/object is placed. This is reflective of the extra-musical theme of information literacy. We are not always able to make assumptions about or discern the source or credibility of the information that we consume. Treatment of space is an essential aspect of the compositional process for electronic and computer music composers.

One has to constantly be aware of the kind of physical space that their music is simulating or distorting – often times compositions sound as if they are continuously augmenting or modifying the space in which the sounds seem to exist. Aside from a variety of panning shapes and techniques I used the Space Designer plugin in Logic Pro X to create the illusion of varied settings for similar material. Space Designer is a convolution reverb, meaning that it uses impulse responses (recordings of the natural reverb of a space) and transforms input signals to make them sound as if they are in such spaces.

Delays

In electronic and computer music composition, delay effects can be used effectively to thicken the texture or to add a sense of distance. Feedback settings, in which the delayed signal is routed back into the input of the module, are also often used to create various echoes and ringing effects. My use of delay lines in this piece is particularly oriented around the creation of clear rhythmic motives. A good example of this can be heard at 7:20 in the recording, the gestures are less frenetic or ambient, there is a clear motivic profile being

developed. Careful manipulation of delay times, manipulating the wet/dry mix of your module and selecting different feedback levels allowed for a sense of relatively predictable meters where the motives could developed. The main tool used for the delay processing in this piece was the GRM Tools Delay plugin. The amplitude and delay distribution sliders are particularly useful for using delays to create rhythms. They specify the amplitude and timing of each delay, respectively, in relation to the previous one. For example, amplitude distribution can be used to create delays that gradually increase or decrease in amplitude (i.e. crescendo), and delay distribution can be used to create motives that are gradually further apart in time.

Fig. 12, GRM Tools Delay– using feedback the series of gradually louder delays will repeat

III. Further Exploration

Although my own tools for granular synthesis worked successfully in generating interesting sound material for my piece, I am certain that further changes can be made to improve their design and overall functionality so that there are as versatile as possible for other computer music compositions and research. I look forward to using them in subsequent pieces to foster a great deal of insight for design improvements.

Another area to be explored is the creation of alternate versions. This piece is intended to be listened to using high-quality loudspeakers or studio monitors, it was not composed or mixed for headphones. It would be ideal to have multiple versions of the piece for various listening environments and speaker setups.

IV. Links to materials for *Sight Unseen*

A .wav of the audio for *Sight Unseen* can be found the on the author's website:

http://www.michaellukaszuk.com/music.html

The code for SuperCollider pseudo UGens, the MIKEGRAN RTcmix instrument and miscellaneous patches can also be found on the author's website:

http://www.michaellukaszuk.com/code.html