Douglas Geers

geersde@music.columbia.edu

Selected Research Projects

Introduction:

For the past several years the primary area of my computer music research has been invesigation into methods of synthesizing dynamic musical textures with the aid of computer technology. My work has examined stochastic procedures, realtime control interfaces, and artificial life most closely. I have both investigated pre-existing computer music methods and software and have designed and written my own. The four examples of my work discussed here are *Appliance*, *Juicer*, *Treembre* and *Ripples*.

I have also completed research into the effects of user interface design on computer music composition, and techniques for building realtime computer music performance instruments. Please email me for more information regarding this work.

1. Appliance: An Interactive Installation Performance

Figure 1: Performance of *Appliance*

In the spring of 2000 violinist Maja Cerar, sculptor Thomas Charveriat, and I embarked upon a collaborative project entitled *Appliance*, which was an interactive performance installation for violin, electroacoustic music, and mechanized sculptures. In performance, the violinist wore a sensor glove to communicate with a Max/MSP patch that controlled a delay line and playback of samples. Meanwhile, she also operated a footswitch that controlled the activities of the sculptures, which were connected into a network (Figure 1).

The sensor glove is worn on the violinist's right (bowing) hand (Figure 2). The glove is handmade from clingy lycra with open fingertips to allow for better

bow control. Three force-sensing resistor (FSRs) are attached to the index, third, and fifth fingers of the glove, and an accelerometer (tracking X and Y velocities) is positioned on the back of the hand. Wires from the glove's sensors and the accelerometer unite at a serial connection which sends the data to a Pic Microcontroller residing in a small switchbox. The chip translates data coming from the glove into MIDI control signals and sends this out to a MIDI interface and the Macintosh computer.

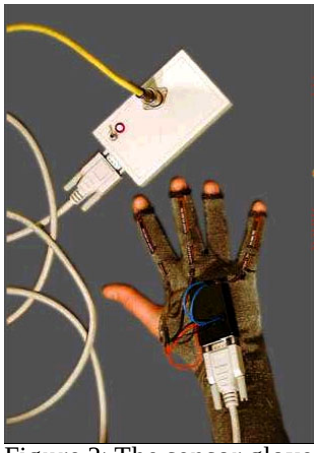

Figure 2: The sensor glove

The MIDI cable feeds into a Macintosh laptop running Max/MSP and a patch built specifically for this piece (Figure 3.) The motions of each finger are mapped to control an individual aspect of performance: delays on/off, samples on/off, and changing the currently active sample bank. The accelerometer data, originally intended to be used to control panning and amplitude, was eventually ignored in performance. The delay time can range from 0.125 second to 5.0 seconds and is controllable from the computer, which is operated by a second performer. The amount of delay feedback is also controlable at the computer. The samples are brief (0.25 to 3.0 seconds) and contain a mixture of violin sounds and other percussive sounds.

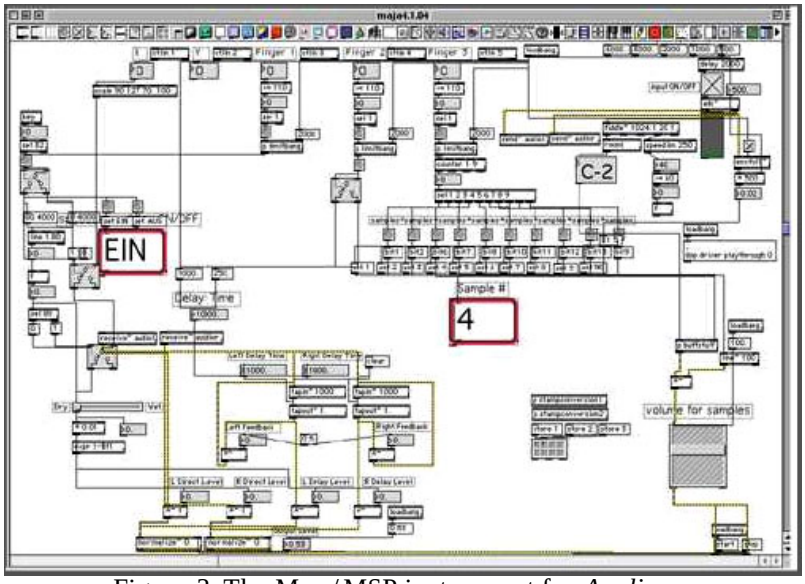

Figure 3: The Max/MSP instrument for *Appliance*

In addition to the controller glove, a small microphone is mounted on the violin and this signal is also fed into MSP. Using the Miller Puckette's `fiddle~` object, we track note onsets, envelopes, and pitch. This data is used to trigger samples when the sampling function is turned on (fifth finger's FSR.) We use the MSP `groove~` object for sample playback, looping short percussive samples to match violin notes' amplitude envelopes and transposing samples according to the estimation of the violin's current pitch.

Each sculpture is composed of motorized elements mounted on a sheet of metal and elegantly laid into a burnished aluminum suitcase. Most of the mechanisms of the sculptures create both motion and sound. Each sculpture also contains at least one microcontroller chip, programmed in BASIC to define the behavior patterns of that sculpture. Because of limitations of both the mechanics of the sculptures and the memory of the microcontrollers, each sculpture's behavior is limited to being a set of loops of rhythmic patterns.

Because the microcontrollers were not designed with musical applications in mind, programming anything more complicated than a metronomic beat becomes quite time intensive. In order for the microcontroller to play a rhythmic patter one must calculate the exact number of milliseconds between each subsequent note onset in the pattern, type in BASIC code using these values for every note of the music, and then load this file into the microcontroller. Obviously, this method of composing musical ideas is quite non-intuitive and can be both frustrating and time-consuming, as the composer cannot hear what s/he has programmed until after the code has been loaded into the microcontroller. To make the process easier, we created interfaces reminiscent of an analog sequencer in Max/MSP to program the sculptures. These allow one to quickly compose rhythms with both auditory and visual feedback. This intuitive interface makes it easy to set up sophisticated polyrhythmic patterns, and then at the push of a button Max writes the BASIC code needed to program the microcontroller.

The sculptures are connected into a network to coordinate their activities. The violinist operates a MIDI footswitch to cycle though a preprogrammed pattern of behaviors of the sculptures. When activated, the mechanism of a sculpture begin to move and make noise. Each sculpture is miked, and this signal is mixed with the violin and the MSP output for amplification.

Each performance of the piece is highly improvisatory. The intention was to set up a system of possibilities for the musician to explore—there is no notated score at all. We made two public performances of the piece, and the second of these was recorded onto video (available upon request.) For those interested, more documentation on this project is available at music.columbia.edu/~thomas, the website of sculptor Thomas Charveriat.

2. Juicer: An Environment for DSP Performance

Currently I am completing a composition project entitled *Gilgamesh.* This work is an hour-long theatrical multimedia work for violin, live signal processing, pre-composed electroacoustic music elements, interactive mechanical sculptures, video, and puppetry. Here I will briefly discuss the aims and general design of the digital audio processing (DSP) system used in *Gilgamesh,* which I have named the *Juicer* (Figure 1).

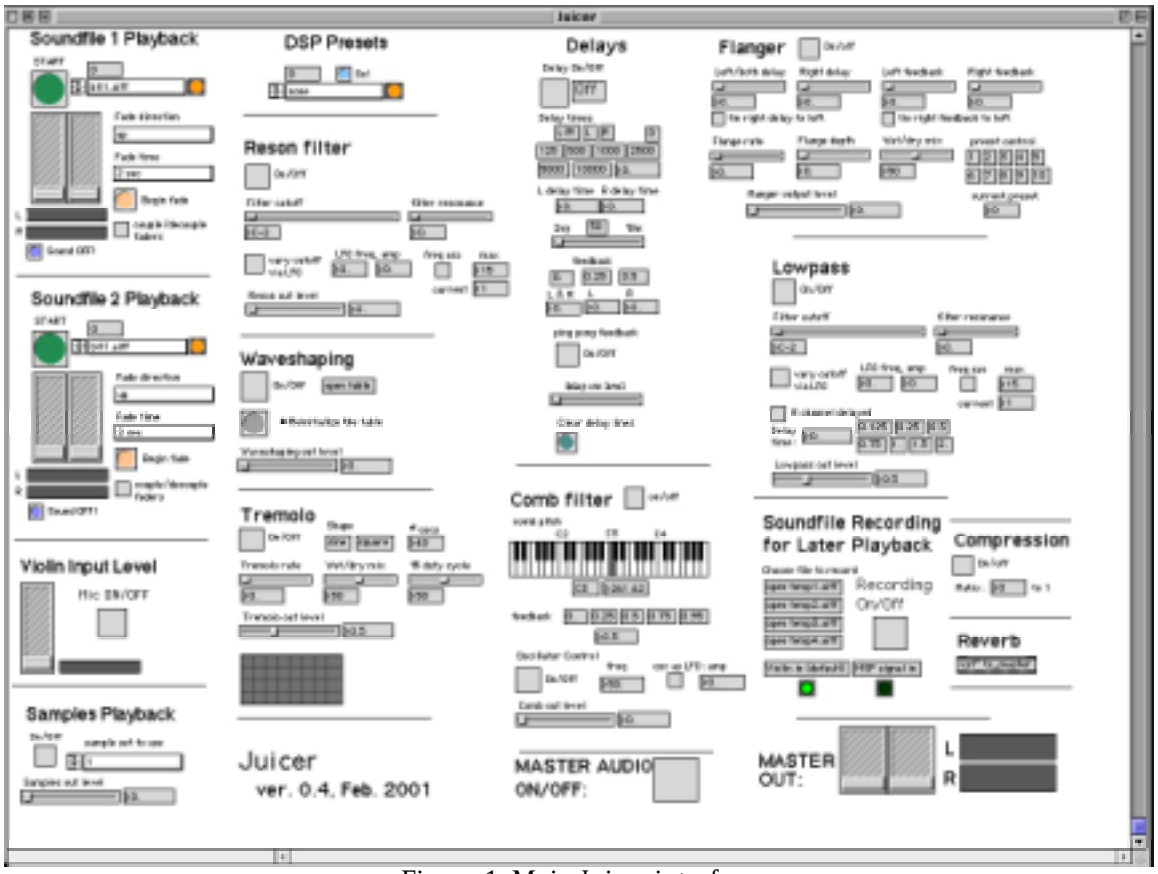

Figure 1: Main Juicer interface

The purpose of the Juicer system is to provide an integrated system for managing the electroacoustic elements of *Gilgamesh.* This includes playback of pre-composed computer music, sampling microphone input for playback later in the work, playing those and other samples in synchronization with microphone input, and management of an array of DSP modules (nine in all). The entire Juicer system was created in the Max/MSP environment.

The DSP portion of *Juicer* consists of (1) nine signal processing modules which are interconnected in a highly reconfigureable way, (2) an interface for adjusting or performing DSP in realtime, (3) a simple method for programming preset combinations of effects and their parameters, and (4) the ability to "morph" from one setting to another over a specified period of time.

Although it was created specifically for performance in *Gilgamesh*, the *Juicer* system was designed in a modular fashion in hopes that many of its elements and possibly the entire system will be useable in future projects. In fact, nearly everything visible on the main *Juicer* interface is the "face"of a separate module, as seen in the example of the flanger instrument below (Figure 2.) In this example, the upper left corner of the patch is the interface, the lower left corner is the system for implementing settings presets for this instrument, and the entire right side is the code for the flanger signal processing itself (with attendant I/O.)

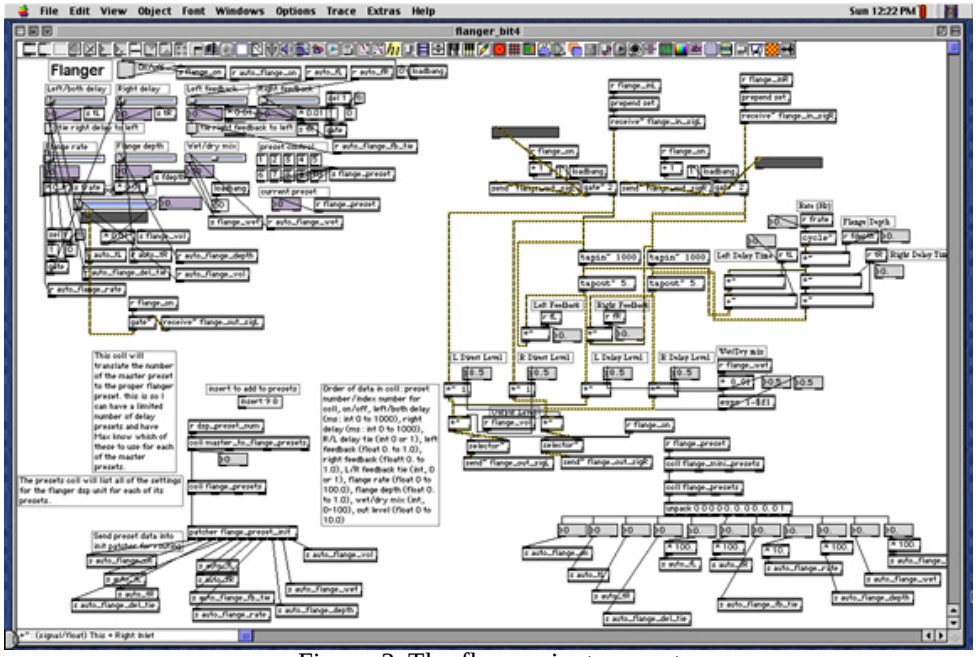

Figure 2: The flanger instrument

Each instrument module of the *Juicer* system was built as a separate Max/MSP patch in this manner. Then each instrument was imported into the final interface via the `bpatcher` object. Since within each instrument module the interface and "engine" of the processing mechanism are separated, it is quite easy for the final interface to present a large, orderly, easily modifiable array of controls without displaying any of the necessary but visually distracting processing algorithms (Figure 1.)

However, although the main *Juicer* interface presents a wide array of sliders, pop-up menus, buttons, and number boxes for activating and adjusting all of the DSP modules, it would be quite a complicated task for one to regularly update the values in each of these in a coordinated fashion during a performance in order to follow a composed score of effects implementation. To drastically simplify this task during a performance, a deceptively simple interface named "DSP Presets" is the facade for a command center that activates preprogrammed effects configurations and controls the simultaneous movement from each preset setting to the next for every instrument module of the *Juicer* (Figure 3.)

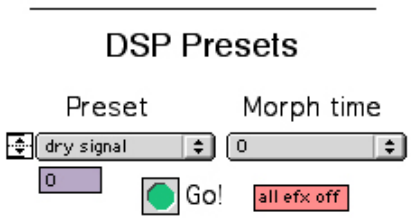

Fig. 3: DSP Presets Interface

The DSP Presets interface has essentially four controls: (1) a number box, increment arrows, and pop-up menu, each of which can increment to the next effects preset from a list created earlier; (2) a pop-up menu to allow the user to select the morph time between the previous preset and the new one; (3) a "Go!" button, which initiates the morph; and (4) an “all efx off” button that acts as a quick DSP mute/reset. In performance, the user simply chooses the next preset, and then at the appropriate moment in the music s/he hits the "Go" button to initiate the smooth morph to the new preset’s effects configuration. Every preset has its own associated morph time, and currently these range from zero milliseconds (meaning no morph time intended) to thirty seconds. In addition, the morph time pop-up menu allows the computer’s performer to overide the preset morph time value if s/he wishes.

DSP Presets operates as follows: Each “master” preset within DSP Presets is associated with data (in a Max `coll` object) which lists the desired preset number of each individual instrument module to use for that master preset. When the user hits “Go” on the DSP Presets interface, it sends each instrument module its destination preset number and the morph time. Each individual module then finds the detailed settings for its new preset in a `coll` object within itself and uses `line` objects to send intermediate values to all of its parameters until their destination values are reached. The rate that the intermediate values are sent is set to match the user’s desired morph time.

Since all of the connections in the chain of processing modules are software links, the order in which the effects are used can change from preset to preset, which is quite satisfying and allows for a wider range of sonic mixtures. One must be careful, of course, when changing the ordering since the sudden re-ordering can cause sudden changes in the signal level going into the currently active modules, resulting in an annoying click; but when managed intelligently this can be avoided.

One limitation of the *Juicer* system is that the configuration data for each preset must be typed into the patch by hand, which is not ideal but can still be done “on the fly” while performing with the *Juicer*. It would be preferable to integrate some kind of `snapshot` element into the system so that the user could add presets whenever s/he found something especially satisfying. However, this has not yet been included.

3. Treembre: A Hierarchical Approach to Sound Timbre Construction

The *Treembre* project is a realtime graphical interface for creating and manipulating sound timbres. Working from the established premise that any complex sound can be dissected into a set of simultaneous dynamic sine waves, *Treembre* gives a sound designer or composer the ability to arrange and manipulate these simple waves within the paradigm of a hierarchical tree structure (Figure 1.) Thus, while each component of the sound retains its individuality, these units may be grouped into larger entities. This grouping allows one to create intelligent relations among the sound's components and manipulate these collections as single regions. The tree-like interface allows the user to quickly and easily create these relations and accomplish complex sound processing tasks, listening to the evolution of the sounds with every adjustment made.

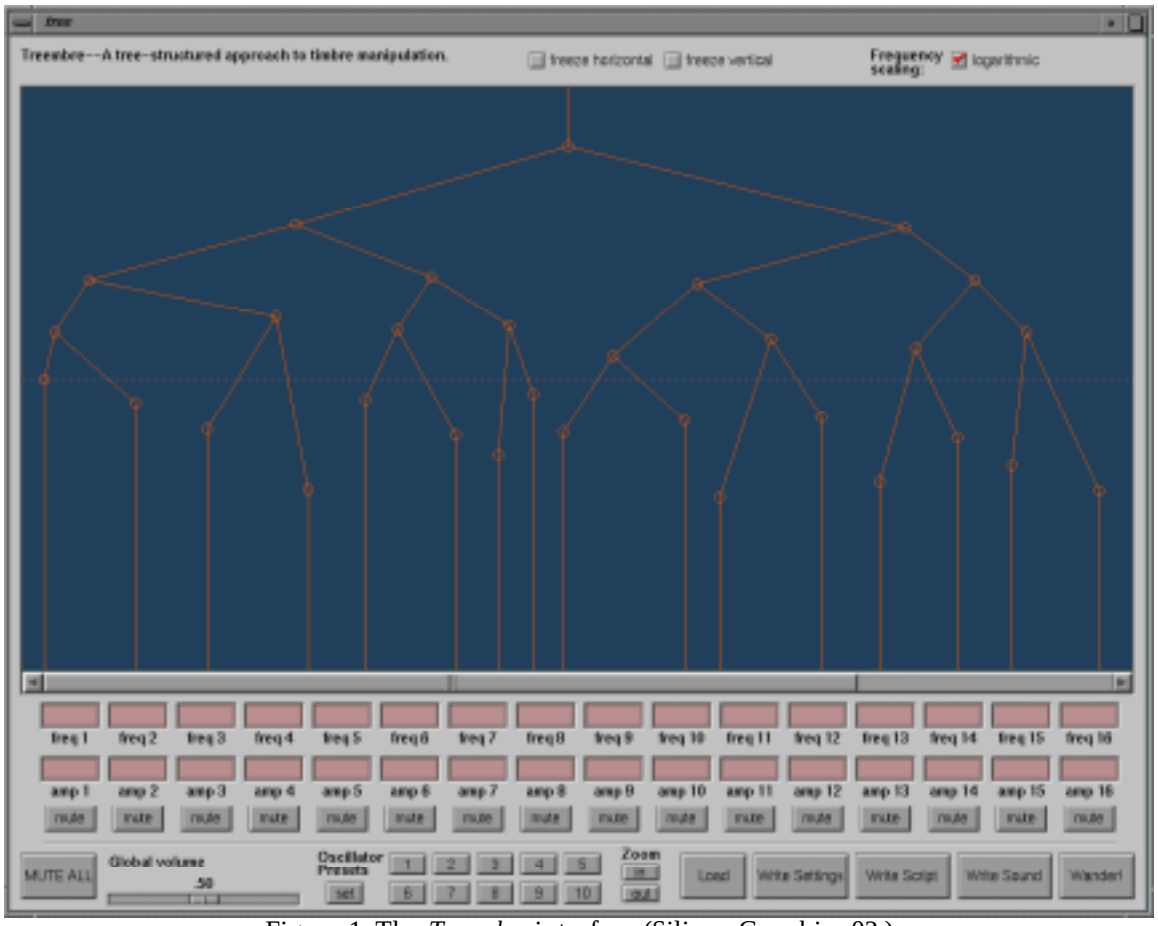

Figure 1: The *Treembre* interface (Silicon Graphics 02)

Treembre was written in C for SGI Irix platform, using XWindows and Motif interface tools. The program generates sound data based upon the location

of tree nodes or "leaves" within a Cartesian plane in which frequency rises from left to right and amplitude rises from bottom to top (Figure 1.) Currently, each node of the tree represents one sine wave component of the current sound. Since complex sounds may be composed of thousands of such waves, my intent is to eventually allow the nodes to represent groups of sine wave components. These groups could be specified by any desired criteria, such as frequency bands, overtone series, amplitude ranges, and so on. Ideally, the user would be able to "open" and "zoom in" to one set of these waves at a time, manipulate it, "zoom out", choose another, and so on. This would combine the ease of the current interface with a more precise degree of control while avoiding the information overload that would result from viewing all of the frequency components simultaneouly.

Another goal is to eventually allow one to create and save multiple configurations and then use realtime controls and/or algrorithmic processes to interpolate among these, creating highly organized and rich sound structures. Given the ever-increasing speed of microprocessors, I believe this is a realistic plan.

In its present form, the *Treembre* application is able to continuously play sound based upon the current configuration of its nodes and will adjust in realtime as any node or group of nodes is moved on the screen. It synthesizes the sound in a two-step process: First it traverses the tree to gather the current data for each node's frequency and amplitude. Then it sends the sound data to *RTcmix*, the realtime sound synthesis and processing environment created by Brad Garton and Dave Topper at Columbia University. *RTcmix* captures the sound data and synthesizes the composite texture using simple additive synthesis of wavetable sine oscillators. Another simple alteration would be to allow for substitution of any desired wavetable shape during sound playback, to enrich the resulting timbre with added partials; but this has not been explored yet.

Originally written on the SGI Irix platform, *Treembre* was ported to the Linux platform by Brad Garton in 1999.

4. Ripples

[Note: This material is an excerpt from a larger paper, "Oblique Strategies," written in 2001 for *Current Musicology*, vol. 67-68 (in press.) Since this article was directed at a musical but non-computer music audience, it exhibits a more conversational style and explains even basic computer music concepts.]

During the composition of *Ripples* I purposely decided to play with two ideas pioneered by composer Iannis Xenakis: stochastic algorithms and a "granular" approach to musical texture (both explained below.) Freely adapting Xenakis' concepts, I decided to create and implement an algorithmic system that would produce music according to rising and falling waves of note densities. Meanwhile, quite unlike Xenakis, I decided to map these ideas to a stable major scale pitch set, clearly pulsed rhythmic material, and a sound world highly influenced by techno music.

Put simply, a stochastic algorithm is a mathematical procedure that utilizes probability theory to influence its decisions. In music, this means that a composer can set up situations that require choices, and allow the computer to make these choices by following statistical distribution rules. For example, one could generate a primitive stochastic melody by instructing the computer to choose every subsequent note via the following rules: 40 percent of the time move down a step, 40 percent of the time move up a step, and 20 percent of the time repeat the same note.

For *Ripples*, I decided to use stochastics to control rising and falling densities of notes in time. From previous experience with computer music I knew that as a regularly pulsed stream of notes gets faster (denser in time), eventually one's mind switches from perceiving it as a series of individual notes to hearing it as a single continuous sound. This notion intrigued me, and I decided to explore perceptions of this boundary in *Ripples*. I conceived of each rise and fall of note densities as a wave of musical energy, and determined that during the piece these would grow and diminish in their extremity; thus the title *Ripples*. As I will explain more below, the final composition was created by connecting and layering many instances of this basic process, run with different input.

I wanted the music in *Ripples* to be pulsed and, at least initially, feel "performable." To accomplish this, I implemented the density changes by creating more and then fewer equal subdivisions ("tuplets") per beat, beginning with quarter notes, then eighth notes, then triplets, sixteenth notes, quintuplets, sextuplets, and so on. After the process reaches its "peak" number of tuplets, it then progresses back from the highest tuplet level to quarter notes once more. Each of these processes, from quarter notes to some x-tuplet to quarter notes again, would constitute one "ripple" (Figure 1.)

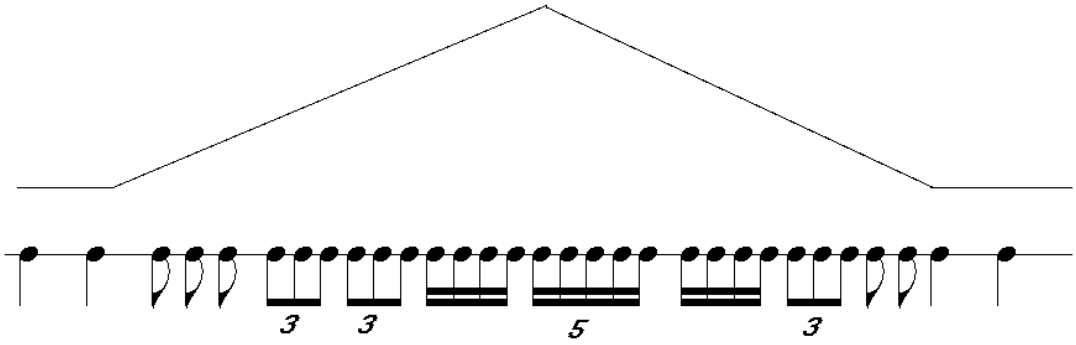

Figure 1: One simplifed, hypothetical instance of the basic *Ripples* algorithm

To realize the *Ripples* algorithm, I wrote my own software in the computer language C that did the necessary calculations and wrote a "score," which I then synthesized into audio using the computer music composition environment. Each time I ran my program, I set values such as tempo, the highest tuplet level to which I wanted the music to progress, a dynamic level for the beginning, a dynamic level to progress to, and the amount of time to get there and back. To give the ripples more shape, I soon altered the original plan so that each "ripple" would contain two density rises and falls: first to the specified peak tuplet level and back, and then to a second peak level and back again. I also added instructions for the music to move back and forth across the left-right stereo field as it played.

Rather than merely having one pulse of notes following this algorithm at a time, I devised my software to create a musical texture of sixteen musical lines, each at a different pitch level, all following the density wave simultaneously but stochastically offset from each other. In other words, I turned my ripple instrument into a sixteen-member ensemble of "ripplers." Moreover, since to me part of what makes instrumental music interesting is the fact that an ensemble of live musicians cannot possibly play perfectly in time together, I employed stochastics so that each rippler would perform slightly differently than its peers. For instance, although technically all the ripplers begin at the same moment, I actually wrote my program so that each one would manifest small random time offsets. In addition, I employed stochastics to individualize each of these ripplers even more, allowing each one to choose its own subset of pitches from the scale (described below), choose a unique time within a specified distance from the "ideal" time to reach its peak tuplet level, and similar stochastic values for the tuplet level to which it progressed, its dynamic levels, and how often it would insert a rest rather than playing a note.

I employed an E-flat major scale as my pitch class set throughout *Ripples*. Over the course of the two rising and falling density waves of each iteration of the ripple algorithm, each rippler chose notes from the scale four times. First, during the acceleration towards the first peak tuplet value, each rippler could choose only two pitch classes. Next, while slowing back from this peak to quarter notes again, each rippler was free to use all seven pitch classes. Then

during the rise to the second peak, each was limited to three pitch classses, and finally during the second fall back to quarter notes each could choose from all seven notes once more. Each time the software chose subsets of the scale it do so entirely at random, though it did check so that it would not choose the same pitch class more than once for each rippler at any given time. Within each segment of music, once a rippler had chosen a set of pitches to use, the software chose from those at random. Thus, although the pitches were those of an E-flat major scale, the use of them was not designed to emphasize tonal relationships. As a result, sometimes one may perceive it as in E-flat major, at other times as in F Dorian, or even at times as C Aeolian.

I mentioned above that each of the ripplers was placed at a different pitch level, and would like to explain a bit further how this was done, and the consequences of my choices. The sixteen ripplers' pitch levels were spaced at the interval of an octave, from the octave C -3 (seven octaves below middle C) to octave C12, eight octaves above middle C. This pitch range may sound a bit crazy, and maybe it is; but it also demonstrates the joys of working with computer music, where one need not limit him / herself to what conventional instruments can do. The lower limit of human hearing is generally between C0 and C1, and the upper limit somewhere between C10 and G10. So my piece was constructed to go beyond these limits, and this produces interesting results, as I will describe next.

Because of the limitations of the 44,100 audio sampling rate I used for *Ripples* and a technicality of digital sound known as *foldover*, all pitches that were supposed to sound from around E-flat10 and above are misrepresented as lower non-tempered pitches (so, no, *Ripples* will not drive dogs crazy.) On the other end of the pitch continuum are notes that are too low for humans to hear; but as these notes play faster and faster tuplets and faster and faster tempos, they begin to be perceived as pitches in relation to the rate they occur (this happens at about 16-20 beats per second.) For instance, if a pitch of eight Hertz (approximately C –1) played sixteenth notes at quarter note=480--which is entirely possible with the *Ripples* software--then a listener would hear a pitch of 32 Hertz, which is very near C1. While I do not have the space to discuss these effects in more detail here, suffice to reiterate that they added interesting, relatively unpredictable pitch and timbral information to the music of *Ripples*.

Beyond this, I let me mention two other ways that I composed the timbre of *Ripples*. First, during my experiments early in the process of composition I intuitively created a particular spectrum that I found pleasing. Later, when the *RTcmix* software synthesized my scores, I instructed it to create each individual note of *Ripples* using this spectrum and an amplitude envelope.

Although my use of one spectrum for all of the notes in *Ripples* might seem simplistic at first, I knew that this would not be a problem because of the second additional way that I compose timbre in the piece. This method is directly linked to the "ripple" process that pervades the piece , and operates as follows: While *Ripples* begins with clear successions of notes, as the music proceeds the basic ripple process repeats and moves to more extreme

realizations—faster and faster tempos. As the density of notes increases, eventually the sense of successive notes, each with its own timbre, dissolves into the perception of a single, complex evolving timbre, now the "line" of the music. In fact, the tempo of *Ripples* eventually rises to a maximum level of quarter note=720, so that even when each of the ripplers is only playing quarter notes it is already performing twelve notes per second!

Music such as this, in which thousands of very brief individual notes are combined linearly to form large scale audio events, is known as "granular synthesis." A simple analogy is that in granular synthesis each note has a role similar to that of each grain on a beach: it is an individual, yet a tiny element of a much larger structure. The concept of granular sound was first employed in composition by Iannis Xenakis, and has been used by many computer music composers during the last quarter of the twentieth century. My interest in *Ripples* was to in effect build a piece around the concept of granularity, allowing the music itself to trace a path from distinct single notes toward an increasingly granular manifestation and back again.

The basic shape of *Ripples* as a whole is the same wave of increasing and decreasing density realized in each individual ripple. As such, the essential slow-fast-slow pattern is easy to discern at the end of the work, and for the most part the musical development within seems smooth and organic. However, while the finished piece seems to flow quite naturally, it is actually the fruit of an involved compositional process.

To create the final composition, I developed my *Ripples* software until it gave me musical output that I found intriguing and satisfying. Then, I ran the program time and time again, altering the settings of the initial values repeatedly. The most significant settings that I altered were the tempo and peak tuplet levels for each ripple, and the piece employs materials at tempos ranging from quarter note=20 to quarter note=720. Each time I ran the program the software realized music within the parameters I had set, using stochastically weighted randomization so that no two runs of the program, even with the same settings, were exactly the same. I listened to these segments of music carefully and adjusted my program settings repeatedly, fine tuning the settings to move the results closer and closer to what I had in mind, until I found just what I wanted, or--sometimes--something even better.

Eventually I was able to create a large number of musical segments embodying a wide range of realizations of the *Ripples* algorithm, and chose to use only those I found most successful from them. I then ordered the chosen segments, edited them, and layered them into a composite mix according to my overall formal plan. I arranged the connections between them so that in the final piece not all of the sections manifest the entire rising and falling density pattern. In fact, most often the connections between sections are quite blurred, and a new iteration of the algorithm, at a new tempo, begins before the previous one has ended. Moreover, as in *Reality House*, sometimes processes are unexpectedly truncated, such as in the abrupt transition back to the opening material at 5 minutes, 35 seconds into *Ripples*. Finally, I processed each segment with several

kinds of audio signal processing software (reverberation, flanging, etc.) to create more and more subtle timbral variations among the sections of the piece.

As the preceding paragraphs illustrate, my creation the *Ripples* software was clearly part of the act of composition, since the design of the program implemented elements of the work's form, its rhythmic and melodic/harmonic material, tempo, number of voices, and the relation of these voices to one another. However, it is also important to note that even though on one level the composition is intensely algorithmic, I exercised "rigorously intuitive" discrimination regarding which materials to choose and how to deploy them in the final composition. Among other things, I chose the tempo settings for each section, the peak tuplet values, the dynamic levels and their changes, the ordering of the sections, how the sections would join one another, when and how much segments would overlap, and how much of the entire sparse-dense-sparse process each segment would manifest in the finished piece.

Another aspect of *Ripples* worth mentioning is the harmonic structure. Essentially, the harmony in *Ripples* is static, employing a single set of seven pitch classes, those of the E-flat major scale, throughout (although as I mentioned earlier, factors such as *foldover* add unexpected pitches.) I composed *Ripples* this way both for practical and aesthetic reasons. First, my software had no mechanism to automatically change the pitch collection. To add planned harmonic changes to the piece would have required either significant further time programming the software or that I run it separately for each harmony, which would have greatly increased the time needed to realize the piece and would have disrupted the continuity of the musical processes. However, a further reason for the static harmony relates to my own interest in musical styles with little or no harmonic motion, including minimal pieces and Indian classical music. Since the concept explored in *Ripples* is the rhythmic/timbral progression, I felt that complex harmonic relations were not necessary and that the relatively stationary harmony was satisfying to me.