2018

Generative Processes for Audification

Judith Jackson
Oberlin College

Generative Processes for Audification

Judy Jackson

April 23rd, 2018

Abstract

Using the JavaSerial library, I present a generative method for the digital signal processing technique of audification. By analyzing multiple test instances produced by this method, I demonstrate that a generative audification process can be precise and easily controlled. The parameters tested in this experiment cause explicit, one-to-one changes in the resulting audio of each test instance.

1 INTRODUCTION

Beginning in the early twentieth century, composers have sought to utilize the sounds of a technologically evolving society in their musical practice. In 1913, the Italian Futurist composer, Luigi Russolo, famously called for the full incorporation of industrialization sounds ('noise-sounds') into classical music. He captured these sentiments and ideas for their execution in his seminal manifesto, The Art of Noise [7]. Russolo's manifesto earned him the title of "the grandfather of noise music" and laid the groundwork for generations of composers to interface with the various screeches and crashes of the future.

This paper explores a new approach to generating complex sonic material for the purpose of composing electronic music. Through the use of the JavaSerial library, composers can craft small modules from data structures and arrange them into various combinations to create larger-scale pieces.

Since 1913, various technological innovations have spurred the birth of a wide variety of electronic music disciplines. The invention of vacuum tubes permitted the creation of smaller, more efficient electronics, and marked the birth of analog synthesizers and synthesized music. Magnetic tape, a product of World War II, allowed composers to capture any sound in the real world and place it in space and time for playback as a musical composition. Likewise, the invention of the computer (and later, the laptop computer), spawned an entire discipline of composition focusing on the creation and modification of digital audio signals. This practice is known as *digital signal processing* (DSP) [7].

The fundamental principle of DSP techniques is the analysis and transformation of five basic parameters of sound: *amplitude, frequency, envelope, timbre*, and *duration*. *Amplitude* describes the intensity or 'loudness' of a sound, and

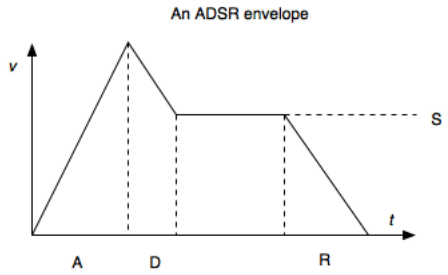

Figure 1: Diagram for an amplitude envelope [16]

uses the unit of decibels, abbreviated dB. The decibel scale ranges from -100 to 100 dB and displays a logarithmic curve. Digital audio representations convert this curve to a 0.0 to 1.0 floating point scale. *Frequency* corresponds to the 'pitch' of a sound. Representing the number of cyclic repetitions per second of a signal, it is denoted with the unit *Hertz* (Hz). Human hearing ranges roughly from 30 Hz to 20 kHz: a concert A4 (the note an orchestra tunes to) is 440 Hz. Any signal lower than 30 Hz is described as *sub-audio*, and is perceived sonically as a rhythm (or clicks at a steady tempo), rather than a pitch. *Envelope* describes the curve of a sound's amplitude, and is broken down into *attack, decay, sustain*, and *release*. Attack refers to the beginning of a sound (the movement from silence to sound), decay and sustain to the middle (the change in amplitude following the attack), and release to the end (the movement from sound back to silence). The description of a sound's envelope is oftentimes referred to as an *ADSR curve* (Figure 1), where A=attack, D=decay, S=sustain, R=release. Some sounds, like the hit of a snare drum, display very sharp envelopes with very short attacks and decays, while others, such as a flute playing a long sustained note, display a smoother envelope with gradual attacks and decays. ADSR curves are analogous to *crescendo* and *decrescendo* dynamic markings in acoustic music.

Timbre describes the quality, or 'color,' of a sound, and refers to its frequency content. The timbre of a sound is what allows the ear to distinguish between a clarinet and a piano playing the same note. A sound's timbre is determined by the number and amplitude of its *overtones*. Each sound exhibits a *fundamental*, or a frequency that the ear determines as the pitch of the sound. In addition to the fundamental, a sound presents a series of overtones (also referred to as *partials*), or frequencies higher and softer than the fundamental that give a sound its color. With more tonal sounds, these overtones fall within the *harmonic series*, a set of frequencies that are integer multiples of the fundamental. Some sounds contain *inharmonic partials*, or overtones outside the harmonic series, in addition to harmonic ones, which give them a 'noisier,' 'diritier,' quality. Timbre, like other sound parameters, exists on a continuum. On one end of the continuum is the *sine wave*, which contains only the fundamental (with no overtones). Because sounds in the natural world always contain overtones, the sine

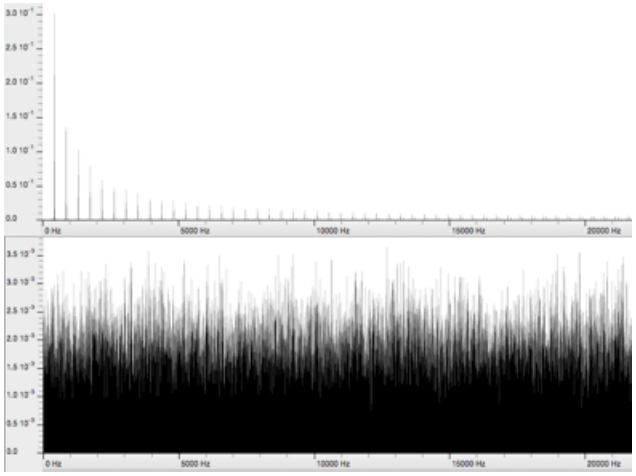

Figure 2: Sonogram for signal with only harmonic overtones (top) and for a noisy signal (bottom)

wave sounds very smooth and alien: the clarinet is its closest approximation in the real world. On the other end of the spectrum is *white noise*, which contains random frequencies at equal amplitudes. All sounds fall somewhere between the two: referring to a sound as "noisy" means that its timbral content contains many inharmonic overtones and that it is aurally similar to white noise (Figure 2). A sound's spectral content is often displayed visually with a *spectrogram* (or *sonogram*), which shows its timbral components and their amplitudes. In this paper, spectrograms will be represented with frequency (in Hz) on the x-axis and amplitude on the y-axis.

Finally, the *duration* of a sound refers to the length of its sounding (the time marking a sound's movement from silence to sound and back to silence). Duration can be described with standard time units (milliseconds/seconds/minutes) or with traditional music notation (eighth notes, quarter notes, etc). All DSP techniques rely on some combination of these basic parameters [7].

Current DSP research divides loosely into three main techniques: *granular synthesis, convolution* and *digital synthesis*. Both granular synthesis and convolution focus on the analysis and transformation of an existing audio signal, whereas digital synthesis seeks to mimic analog synthesis by performing the combinations originally done with electrical oscillators within a computer.

The technique of *granular synthesis* relies on the division of a digital audio signal into small pieces, or *grains*. These grains are then transformed and recombined in various ways (Figure 3, 4). Research on this technique focuses both on the modification of individual grains and the methods used to re-synthesize them into a continuous signal. The former approach allows composers control over parameters such as the amplitude, timbral content, and envelope of each grain. Grain-level processes can be uniform (each grain goes through the same

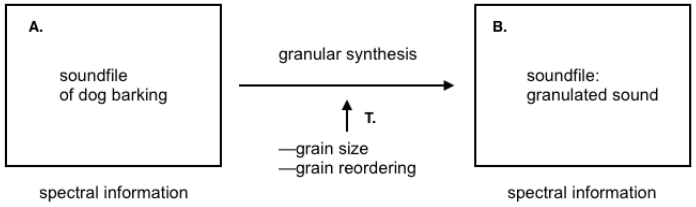

Figure 3: The process of granular synthesis

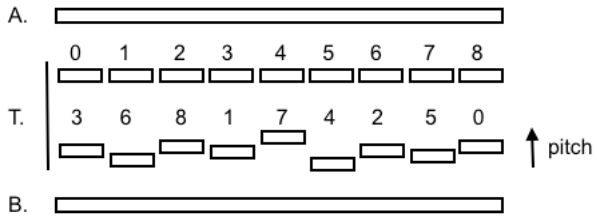

Figure 4: The process of granular synthesis

transformation), or specific (each grain receives a set of parameters unique from the others). Oftentimes, composers apply aleatoric processes to the latter approach: for example, a random number generator can determine the modified amplitude of each grain. When re-synthesizing the grains, composers control parameters such as grain order and playback speed [11].

Current DSP research continually presents novel approaches to grain modification and recombination, but the sonic results of these techniques always sound like granular synthesis. While the computational feat involved with executing the new approach may hold merit from a scientific standpoint, its artistic results fail to offer composers new sonic material with which to compose.

The technique of *convolution* involves the application of the *Fast Fourier Transform* (FFT) onto an audio signal. Like any complex signal, an audio signal can be reduced to a series of sine waves of various frequencies and amplitudes. The amplitudes and frequencies of these sine wave components provide composers with a sound's *spectral components*, or the set of data describing its timbre: the combination of these frequencies at these amplitudes are what make the sound of a dog barking different from the sound of a car driving by [11].

The data provided by FFT analysis acts as the foundation for an entire host of computer music techniques: *reverb* (the sound of an object in a specific space), *filtering* (a technique that allows only certain frequencies to pass through), speech synthesis, and *cross synthesis* (transforming one sound into another) all rely on FFT for their execution (Figure 5). Because of this, all of these techniques can be grouped into one subset of DSP research, which will be referred to as convolution in this paper. The basis of convolution techniques is as follows: a composer analyzes a sound using FFT, modifies its spectral

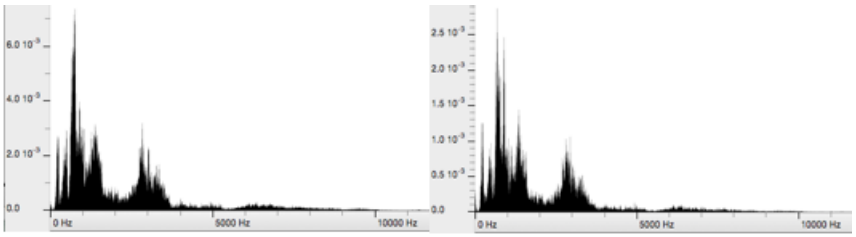

Figure 5: Spectrums for a sample before (left) and after (right) applying reverb

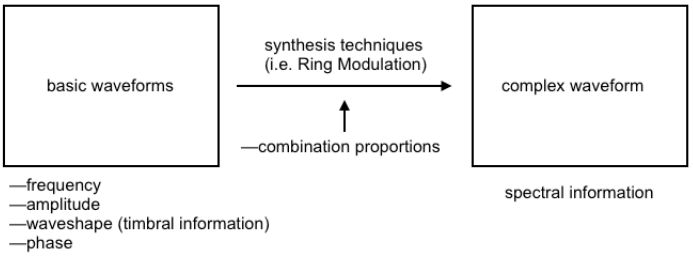

Figure 6: The process of digital synthesis

components through deletion, amplification, attenuation, or addition, and re-synthesizes it to produce new sonic material. This method can be used to place an existing sound in a new space (i.e. one can make the sound of a dog barking into the sound of a dog barking inside a cathedral), to time stretch a sound without changing its pitch (one simply extends the durations of each of the partials to the desired amount), or to place a new sonic envelope onto an existing sound (i.e. one can make a wind sample sound as if it is saying words like a human), to name a few [11]. However, like granular synthesis, convolution techniques display a distinct sonic flavor. Any sound modified by this technique sounds like a convolution transformation–regardless of how novel the technological approach used to create it, it remains within the same sonic palette.

Digital synthesis seeks to recreate the technique of *analog synthesis* within a computer (Figure 6). Analog synthesis creates sound by using mathematical processes to combine basic waveforms. Analog synthesizers contain electrical oscillators that produce some combination of sine, triangle, square (or pulse), and sawtooth waves. These basic waveforms are named for the shape they produce when the oscillator's current is displayed visually with a oscilloscope.

Each of these basic waveforms possess known overtone structures (Figure 7) that make their combinations predictable and standardized. Some of these techniques include *frequency modulation, amplitude modulation, ring modulation*, and *filtering*. These processes are easily identifiable by a trained ear and can be recreated from synthesizer to synthesizer. In digital synthesis, these waveforms are created by plotting their phase, shape, and period to a bitstream.

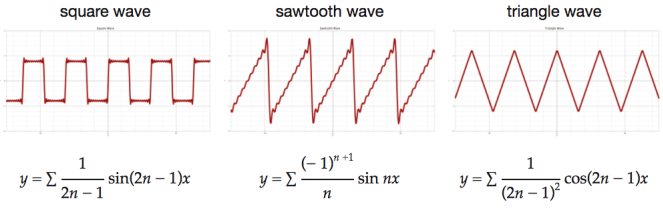

Figure 7: Basic waveforms and their overtone structures [11]

Standard synthesis techniques can then be performed on these bitstreams using basic computational procedures (for example, ring modulation is performed by multiplying the two signals together) [3]. As a result, composers can control these techniques easily and precisely.

Each of the aforementioned techniques is backed by a supple body of academic research. The composer looking to apply and extend these techniques can access a multitude of resources that explain their technological infrastructure and the manners in which they can be manipulated to produce known sonic results. However, recent research has focused mostly on redesigning and extending these techniques. As a result, little effort has been put into discovering new methods of manipulating digital audio by the academic community.

This lull in innovation in academia stands in stark contrast to the work being done by composers in non-academic, underground electronic music scenes. Many of the software programs designed by institutions end up in the hands of experimental musicians, who push the programs past their capabilities in order to create new sonic material. Musicians refer to this genre of music by many names, one of which is *glitch audio* [4].

Glitch refers to a variety of music making techniques, such as misusing noise-reduction software, reading non-audio files (such as *.pdf* and *.txt* files) as audio, and causing compact discs to skip. While the methods vary, the purpose behind them unites the genre: glitch artists force technology to fail and capture the results in order to produce art. Glitch techniques tend to rely heavily on empirical methods: Artists try different combinations until they find results they like. As a result, the artist lacks information as to how and why a specific result occurred, and little research exists to explain the scientific and computational underpinnings of this practice [4].

This paper explores one technique of glitch audio that involves reading binary data as uncompressed audio files (*.wav* files). Dombois and Eckel define the term *audification* as "interpreting any kind of one-dimensional signal as amplitude over time and playing it back on a loudspeaker for the purpose of listening" [5] and I will refer to this technique and the sonic results it produces as data audification and audifications, respectively. Data audification is commonly confused with a practice known as data sonification. Data sonification, however, refers to the process of mapping a data set to predefined sonic events

or parameters, and requires a composer to decide how variance in the set affects the sounds or parameters over time. Data audification is simply the process of reading any file as an audio file; the content of the file still affects the sonic result, but the composer has no choice over the result aside from choosing which files to audify.

An understanding of how digital audio is stored and played back by a computer is required in order to better understand the computational processes in action behind audification. A common file format for storing uncompressed audio data is the *.wav* file. For the purposes of this paper, all audifications created and analyzed will be stored in *.wav* format. A *.wav* file consists of a header, which contains the format specifications of the file, and a 'data' section of variable length, which relays the sonic information of the file. The 44 bytes of the header contain information identifying the file as a *.wav* file, as well as the *sampling rate, bit depth*, and number of channels of the audio relayed by the data section. The *sampling rate* of the file determines the number of data points in a second of audio, and the *bit depth* determines the amplitude resolution of the signal. The *digital to analog converter* (DAC) reads the data section of the *.wav* file according to the method outlined in the header. Given a mono *.wav* file with a 44.1kHz sampling rate and signed 8-bit bit depth, the DAC constructs a waveform by reading 1 signed, 8-bit value at a time, in little endian form, when using PCM encoding, from the data section, and plotting 44,100 values per 1 second of audio. Likewise, the DAC converts files with bit depths of 16-bit, 24-bit, and 32-bit float by reading in 1 signed 16-bit short integer, 1 signed 24-bit integer, or 1 signed 32-bit floating- point value at a time (Figure 8). Files with higher sampling rates contain a larger number of data points per second of audio, so they create higher resolution waveforms and higher fidelity sound. Files with more than one channel interleave the data for each channel: with stereo files, value $a[i]$ corresponds to the left channel, value $a[i+1]$ to the right channel, value $a[i+2]$ to the left channel, and so forth. For the purpose of this paper, analysis of the digital to analog conversion process will be based on mono, 44.1kHz, signed 16-bit PCM *.wav* files [12].

Many digital audio editing programs, such as Audacity, Amadeus, and SoX, contain a *raw data* function that allows the user to read any file on their computer as an audio file. The raw data function treats the given file as a *.wav* file without a header: The user manually inputs values for sample rate, bit depth, and encoding. Then, the program uses these specifications to generate a waveform from the values of the file as it would with the data section of a *.wav* file. This method produces unique sonic results with high levels of timbral and rhythmic complexity, and are unlike those generated by standard synthesis or DSP processes. Specifically, they demonstrate an irrationality in timbral content and rhythm that differentiate them from the mathematical products of standard synthesis. The correlation between binary code segments and their audifications becomes more clear through analyzing the underlying mathematical structure of the digital to analog conversion process. Consider a file F containing n repetitions of a byte sequence $S = b_0, b_1, ..., b_i$, where $b_j, 0 \leq j \leq i$ is one byte. The file contains only unvaried repetitions of S and therefore represents

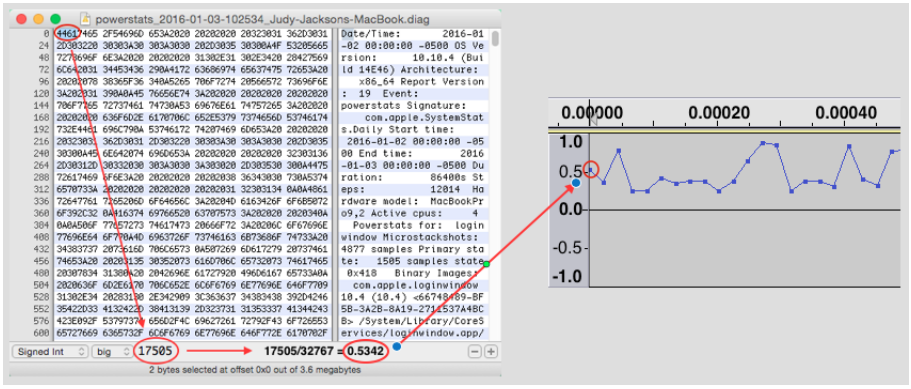

Figure 8: The process of audification

a periodic function. Thus, when audified, it produces a tone with consistent frequency and timbre. As a result, we can analyze the file and its audification like any other periodic function (i.e. the sine function). The frequency, represented in Hz, of the resulting tone can be calculated using the length S in bits, a sampling rate, R, and a bit depth, D:

$$Hz = \frac{R * D}{|S|}$$

The byte content of S directly affects the timbral content of its audification, albeit in a less mathematically explicit way. As previously mentioned, the DAC derives the shape of the waveform by plotting each sample x at the sample rate, with each point being $|x| = D$ bits long. Reading a sample x_k as a numeric value, whose type is specified by D (i.e. reading a sample as a signed, 8-bit short with little-endian encoding), returns the value of the waveform at index k. Thus, a sequence's byte content affects the timbre of its audification.

If S is too long or too short, the audification will result in a sub-audio frequency or a frequency too high for human perception (i.e, given 44.1kHz, 8-bit specifications and a 30Hz-20kHz range of human hearing, S must be between 2.2 bytes and 1470 bytes long). Likewise, if the sample values calculated from S are too random (especially if S is long), individual instances of S become indistinguishable and the file loses its periodic characteristic. As a result, the audification of the file produces a sound that is more noise than tone. If one generates a binary file using a random number generator, the audification of the file will be white noise. In fact, generating white noise for DSP system testing relies mainly on random number generators [6].

Finally, the size of the file L, in bits, directly corresponds to the length T of the audification in seconds. Again, this can be demonstrated mathematically, given a sample rate R and a bit depth D:

$$T = \frac{L}{R * D}$$

Using this knowledge, one can begin to explore how existing file types manifest as audio. Understanding the audifications of standardized file formats lays the groundwork for applying this knowledge to generating new binary sequences in order to create new sounds.

An analysis of various file types and their binary structures reveals that the format specifications directly influence the audification of a file. Simple structures, such as *.txt* files produce simple audifications, and more detailed formats, such as *.psd* files produce highly dynamic, complex sonic products.

Non-musical files are file types intended for functions other than storing audio data. These include, but are not limited to, files storing image data (*.bmp, .jpeg, .tiff*, etc.), files storing text data (*.txt, .pdf, .docx*, etc.), files storing executable code (*.exe, .maxpat*, etc.) and files containing data about a device or computer (*.log, .diag*, etc.). I examined *.txt* (text files), *.bmp* (uncompressed bitmap image) files, and *.psd* (Adobe Photoshop) files specifically. File format specifications designate the byte content and structure of a file. Files of the same type are outlined by the same file format specification. Each file type follows a structure that remains more or less consistent between instances of that type. For example, every *.wav* file has a 44-byte header followed by a variable-length data section, and begins with the file signature for a *.wav* file [12]. Consistent structuring of a file type manifests itself in the audifying process. This means that files of the same type will sound more similar to each other than files of different types. The relation between a file's format specifications and its audification can be demonstrated through an analysis of the structures of three file types: *.txt* files, *.bmp* files, and *.psd* files.

A simple example of the correlation between file types and their audifications is the *.txt* file (text file). A text file contains no headers and its binary data is simply an ASCII encoding of its text content (each character is represented by a value from 0-127). Therefore, the text content of the file directly impacts its audification. Consider two text files with different contents: File A contains a large, alphabetized word list, and File B contains the entire text of Jane Austen's *Pride and Prejudice*. The content of File A is structured because it contains sections of variable-length words that start with the same character and are delineated with a space. The structured nature of the con- tent creates a slightly rhythmic audification. In comparison, File B contains a more uniform distribution of characters, making its byte sequences more random. As expected, the audification of this file is simply noise (Figure 9).

In contrast, uncompressed *.bmp* files (image bitmaps) exemplify a more structured and complex file format. The structure of a *.bmp* file can be broken down as follows:

–a file header (14 bytes) identifying the file as a .bmp

–an image header (40 bytes) that specifies the dimensions, format (bits/pixel), and compression method of the image (this analysis examines uncompressed

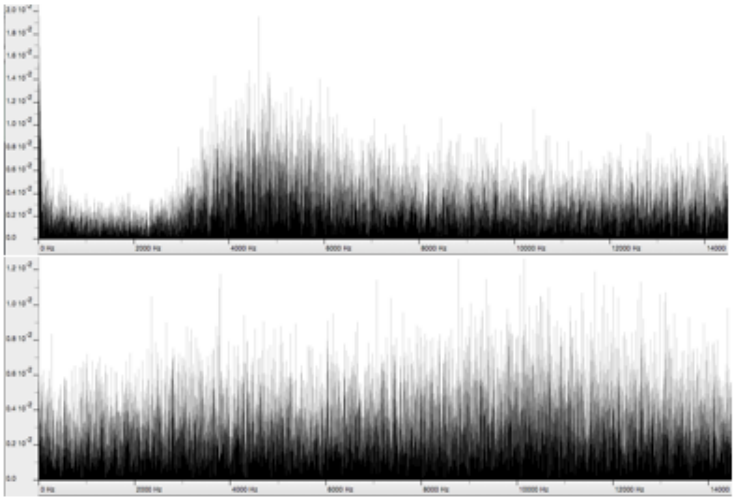

Figure 9: Spectrograms for *.txt* of large dictionary (top) and for *.txt* of 'Pride and Prejudice' (bottom)

.bmp files)

–a variable-length color table (not always present) that provides a lookup table matching pixel values to colors

–the pixel/image data (variable length), which lists values pixel by pixel, row by row, top to bottom. In uncompressed *.bmp* files, the pixel value is equal to its RGB value. Each row of pixel data begins with a double word (4 bytes), which distinguishes rows from each other [14]

The relatively small byte sizes of the headers make their impact on the audification of the file negligible. This means that the resulting sound essentially reflects the content of the pixel data section. Audifications of *.bmp* files produce tones with consistent frequencies: the length of the pixel rows (the width of the image) determines the frequency of the tone. In this case, the row acts as the repeating byte sequence, and distinguishing individual sequences is made possible by the double word beginning each sequence. The values of the pixels in the row (the RGB value) determine the timbre of the resulting tone, and the height of the image determines the duration of the tone (due to the fact that the height determines the number of rows, or the number of repetitions of the sequence). This behavior becomes apparent when comparing a control *.bmp* file to other *.bmp* files that vary in color, image width, and image height when compared, as seen in Figure 10.

Adobe Photoshop files (*.psd* files) rely on a file format that presents an even higher level of structure and complexity. An analysis of six *.psd* files containing black and white animations demonstrates the correlation between

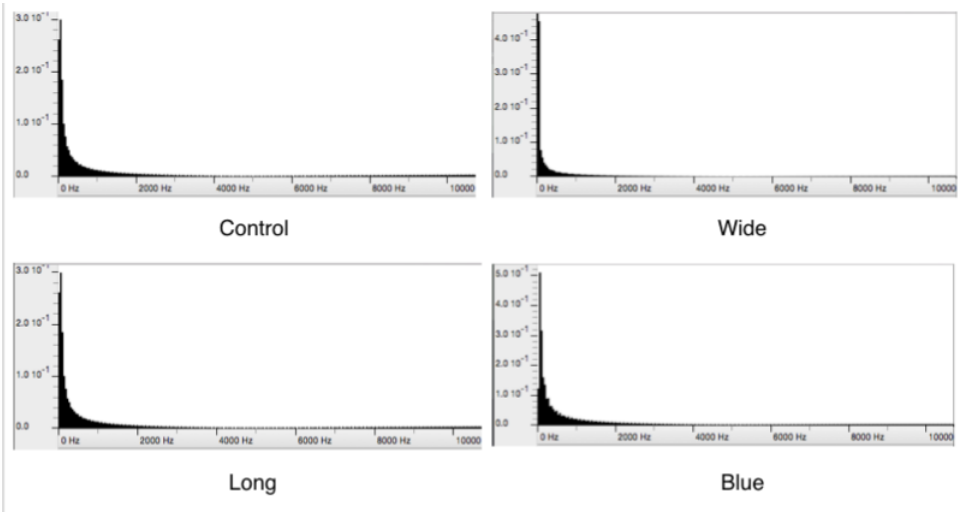

Figure 10: Spectrograms for different .bmp files

the structure of *.psd* files and their audifications. Although the six files varied in size and subsequent audification duration, each demonstrated a consistent sonic structure from file to file. The file format specification for *.psd* files reflects this consistency. Each *.psd* file follows the general structure outlined in Figure 9 [9]:

The 'Layer and Mask Information' section contains two subsections, 'Layer Records' and 'Channel Image Data', that produce structurally important components of the audification. Using the details in the *.psd* file format specification, one can map the byte structures contained in the file to sonic sections in the audification. Each of the audifications of the six files contained three distinct sonic sections:

1. Section 1 consisted of a timbrally complex, high pitched tone that varied slightly in frequency and noisiness. The tone exhibited a rhythmic beating that varied slightly in tempo. An analysis of the parsed .psd file revealed that this section corresponded to the 'Layer Records' portion of the file's binary data. Photoshop allows a user to construct an image with 'layers'. The binary data in this section provides an overview of each layer, such as specifications for the number of channels, the masking parameters, and the blend data in the layer. The overview for each layer is relatively rigid in terms of its byte size and composition. Because each layer contains more or less the same amount and type of binary data, the audification of this section produces a tone with relatively consistent frequency and timbre, with slight fluctuations. Each layer's overview is padded with null bytes (0x00) at the end for rounding purposes. Null bytes produce silence when audified (the resulting waveform has 0 amplitude), so the padding produces the rhythmic effect present in the audification of this section. The slight variance in the tempo of the rhythm stems from variations in the size of each layer data sequence and the amount of padding present in

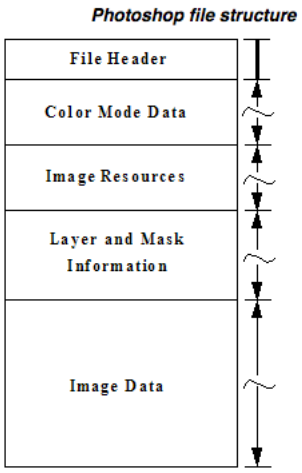

Figure 11: Outline of .psd file format

the sequence.

2. The second sonic section featured narrowly banded, high frequency noise gestures that appeared in tuples of three. These gestures varied in pitch, noisiness, and duration from grouping to grouping, but gestures within each group of three demonstrated the same sonic characteristics. This sonic section corresponds to the 'Channel Image Data' portion of the *.psd* file. The amount of image data varied from layer to layer, causing the tuples to vary in pitch and duration. Each gesture in the tuple corresponds to the R, G, and B data for each layer, hence their appearance in threes in the audification. The original Photoshop files were black and white, so the R, G, and B values for each layer were all the same and created the consistency in timbre within a tuple. The ratio of black to white in the original file varied from layer to layer, causing the variance in timbre between tuples.

3. The final sonic section featured three gestures similar to a tuple created by the Channel Image Data section. However, these gestures had significantly longer durations (about 3 seconds/gesture, compared to milliseconds/gesture in the Channel Image Data section). This sonic section corresponded to the 'Image Data' portion of the file, which contains the RGB values for the entire file. Again, the *.psd* files in question are uncompressed so the data in this section is stored in planar form: All the R values of all of the layers are followed by all the G values for all of the layers, which are followed by all the B values for all of the layers in the file. Because the original files were black and white, each gesture contained the same byte content. Therefore the R, G, and B byte sequences in this section produced three gestures of identical duration, pitch, and timbre in the audification.

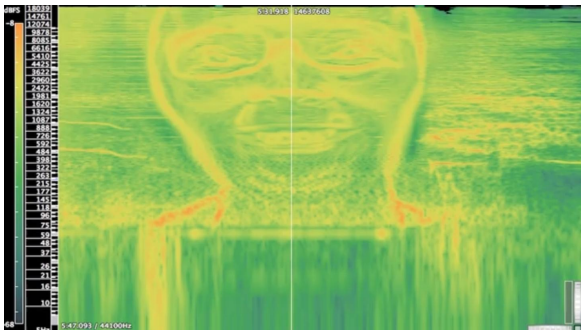

Figure 12: Sonogram picture found in Aphex Twin's 'Equation' [1]

To summarize, similar byte contents and structures in files of the same type create similar audifications. Files with higher levels of complexity and structure, such as the *.psd* files, produce more complex audifications, whereas simpler formats, such as *.bmp* files, produce simpler audifications that are tones instead of complex gestures. These conclusions prove helpful when designing a generative approach to glitch audio. In order to create binary files that produce compelling audifications, such an approach must generate structured, slightly varied repetitions of binary sequences that have some degree of complexity. The JavaSerial library allows users to write any data structure in a Java program to a binary file, thus meeting all the requirements necessary. The experiment described in this paper demonstrates the execution of this method and its application towards generating innovative sonic material.

2 RELATED WORKS

A number of experimental electronic pop artists have incorporated glitch techniques into their practice by inserting image data into their music. These tracks are commonly referred to as "easter eggs." Several online forums dedicate themselves to finding these extra-musical components in artists' releases. One such example of this technique is Aphex Twin's *Equation* (Figure 10).

By viewing the sonogram (a visual representation of the audio file's timbral content), one discovers an image of the artist's face [1]. Other examples include the sonogram of Venetian Snares' Look, which reveals a picture of the artist's cats [10], and Plaid's *3recurring*, which contains an image of repeating "3"s [15]. While these artists all incorporate image data into their products, none of these tracks actually directly utilize the binary data of the image files. Instead, the artists use software programs, such as MetaSynth, to convert an image into spectral data (partial frequencies and their amplitudes) rather than audio sample values. Additionally, the artists employ this technique to add an extra layer of artistry to their music rather than to capitalize on the sonic results of the process.

Some contemporary classical composers have investigated DAC conversions of binary data. However, their work mainly focuses on the aural quality of the sonic result rather than the scientific relationship between the data and the DAC's output. In his seminal piece, Riverrun, Barry Truax implemented a bitstream conversion program to perform granular synthesis, a technique that breaks a signal into user-specified sized grains, in real time [17]. However, Truax's implementation aimed to create a new electronic performance technique, and focused on the aesthetic quality of its results rather than the methodology behind it. Therefore, little analysis exists as to how the input bitstream affects the resulting grain output. European serialist composer GM Koenig also manipulated binary data for sound production in his SSP [Sound Synthesis Program] research. Koenig's program produced its output by generating waveforms from time/amplitude value pairs and transforming them through various operations. Koenig's program provides some analytical foundations: the composer describes the output not with traditional frequency and timbre specifications but with a list of the operations performed on the data. Berg et. al, in their paper about Koenig's SSP program, note that when certain parameters are changed in specific ways, the resulting outputs demonstrate consistent sonic tendencies (i.e. setting the `GROUP SIZE` parameter to a large value concentrates the wave's partials around the lower frequencies) [2]. While there exists more analysis of the relationship between the input and output of SSP, these conclusions were reached through testing and analyzing multiple sample outputs and fail to establish some generalized scientific theory relating waveshape to timbral content.

Composer Iannis Xenakis also explored value-based waveform generation in his GENDY program. The sonorities produced by this program, specifically those in his piece *S.709*, demonstrate a striking resemblance to the audifications of Photoshop files. The program dynamically generates and modifies a waveform by applying Markov chains to user-selected breakpoints in the signal. This process causes slight deviations in both the length and the shape of the waveform. As a result, the program produces an output with a wavering timbral quality and slight frequency oscillation [13]. Like Truax's implementation of granular synthesis, Xenakis' program focuses on the generative and composerly aspects of the process, and lacks a thorough method of mapping the program's manipulations to its output. Overall, little research exists as to how a file's byte content affect its audification and, more generally, how a wave's shape corresponds to its timbral content. This work attempts to fill in some of those gaps.

3 METHODOLOGY

One shortcoming of current audification processes is the lack of user control over the result. While a composer can choose which files to audify, manipulating the file's content in a meaningful and exacting way requires the tedious process of editing the byte values of (sometimes very large) binary sequences within a hex-

code editor. In choosing which files to audify, composers can develop intuition as to which files produce meaningful results and which do not, but learning these nuances requires audifying a large variety and volume of files. Composers can learn which files produce what types of sounds, but understanding why these results occur remains somewhat elusive.

Approaching audification from a generative angle can offer a solution to these problems. By creating their own structures, composers can control exactly what a file contains. Editing these files becomes easier: one can simply change a single parameter within a script to make large-scale changes (as opposed to making these changes by hand in a hex-code editor or through composing a new regular expression each time an edit is required). Additionally, audification scripts can be reused for different pieces, streamlining the compositional process

The JavaSerial library offers an appropriate platform with which to execute this process. The library allows users to write data structures directly to disk, making it ideal for creating binary files to read as audio. The library is also standardized: common data structures (such as trees), will always be represented using the same binary code, allowing for consistency across program runs. Thus, if composers wish to change only the content of their structures (i.e. change a tree's content from integers to floats), the overall framework of the structure will stay the same, allowing composers a degree of precision in their construction process. This standardization also allows for consistent representations across iterations: if a composer decides that they need two more seconds of an audification of a repeating structure, they can simply write more iterations of the structure to disk without changing the sonic palette of the audification.

Having determined that some sort of generative process within audification may assist composers looking to utilize the technique, we must begin to explore how to approach this method with control and understanding.

To begin an examination of what a generative audification process may consist of, I designed a small, nested data structure (Figure 11). This structure (which I will refer to as a *nest*) consists of trees containing random shorts (within a specified range), stored in a linked list, with several of these linked lists stored inside a larger linked list.

The design of this data structure factored in the following parameters:

1. The structure must have some degree of complexity (not just a simple repeating byte sequence), but not so complex that the relation between it and its audification becomes obscured. My analysis of existing file formats and their audifications informed this requirement. Files with more complexity produced more musically interesting sonic material. However, small changes in highly complex files produce large changes in their audifications, making user manipulation difficult. However, simpler files demonstrate clear correlations between file format and audification, and leave few parameters within the file to manipulate. Thus, a generated data structure must have enough complexity to produce complex audifications with a variety of possible parameter manipulations, but not so much complexity that the user lacks control over the sonic results.

2. The size of each layer must be easily controlled. As explained, the size of

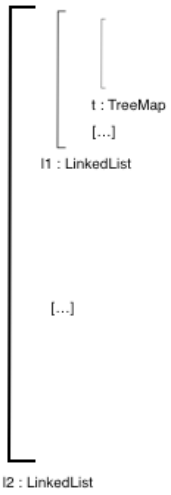

Figure 13: Outline of 'nest' data structure

a repeating byte sequence directly affects the frequency of the audification, so control over this parameter is necessary for control over the structure's audification.

3. The content of each layer must be easily controlled. Again, the content of a repeating sequence determines the waveshape (and thus the timbre) of the audification, so control over the audification necessitates control over the structure's content.

I controlled the following parameters within a nest in order to illuminate how modifications to the structure affected aspects of its audification:

1. *Container size*: The size parameters correspond to the number of iterations of each nest component. The number of nest iterations written to disk remained fixed, due to the fact that changing this parameter simply changes the length of the audification. The size parameter has 3 possible values, corresponding to the number of iterations of a component: 20 (small), 50 (medium=default), 100 (large). When analyzing the effect of component size on audifications, only one components' size was changed, while the other two components remained fixed at the default value of 50 (i.e. `T_SIZE`=20, `L1_SIZE`=50, `L2_SIZE`=50).

a. `T_SIZE` : the size of the TreeMap (the number of nodes stored in each tree)

b. `L1_SIZE` : the size of the internal LinkedList (the number of TreeMaps stored in each internal list)

c. `L2_SIZE` : the size of the external LinkedList (the number of internal

LinkedLists stored in each external list)

2. *Random seed*: a seed for the random number generator in Java. Including control of this parameter allows for consistency across runs of the program. I used 255 as my random seed for each generation.

3. *Random range*: the range of random numbers generated to populate the trees. Within a nest, each tree contains the same values (i.e. $L1[x]$ contains the same values, in the same order, as $L1[y]$). The two aspects of random range that I examined were bandwidth and location. Because each component has a fixed structure, varying the values contained inside of them provides the timbral variation of the audification. I chose to use random numbers instead of numbers with some sort of pattern in order to maintain the spirit of glitch: randomness correlates with noisiness and makes the audification more dynamic.

a. *Bandwidth*: How wide the random range is (`RAND_HIGH - RAND_LOW` = bandwidth). I fixed bandwidth to three possible values: wide=10,000, mid=5000, narrow=1000). Because each tree contains shorts, which can have values from -2^15 to $2^15 - 1$, these bandwidth values allow varying degrees of randomness while maintaining ranges with values all below zero, with a zero crossing, and all above zero.

b. *Location*: Where the random values fall. Low: [`RAND_LOW` $= -2^{15}$, `RAND_HIGH` $= -2^{15} + 1 + bandwidth$], Mid: [`RAND_LOW` $= 0 - (bandwidth/2)$, `RAND_HIGH` $= 0 + (bandwidth/2)$], High = [`RAND_LOW` $= 2^{15} - bandwidth - 2$, `RAND_HIGH` $= 2^{15} - 2$]. For the upper and lower extremes (`RAND_LOW` for low range and `RAND_HIGH` for high range), I moved the limits towards zero by a value of 1 to avoid aliasing issues when rendering the audification. When an audio file experiences values at the extremes of the 16-bit range, frequency artifacts can appear and obfuscate the results of FFT analysis.

With each set of parameters, I decided on three fixed values to limit the number of files examined. Audifications with fifty trees will not differ greatly from audifications with fifty-one trees, therefore using 50 as a generalized "medium" size will capture the effect of size on the resulting file. I applied this rationale to the random parameters as well. Using these variable sets, I generated a file for each permutation of parameters, totalling in 72 total files (see Results section for full chart).

Taking these files, I created audifications of each with the program SoX, an audio editing software executed via the command line. I then listened to each audification to observe general "musical" characteristics of each (such as rhythm and similarity to other audifications). I then performed FFT analysis using Amadeus to examine how these characteristics corresponded to the structure and composition of the files. With each FFT analysis, I used a 4096 bin size in order to have precision in the frequency-domain values.

4 RESULTS

The audifications produced by each test file can be separated into two groups: 1) files that demonstrated a rhythmic oscillation between a steady tone and

noisier material [Group I], and 2) files that manifested as noise with no clear rhythm [Group II] (Figure 14).

These two groups derive from each file's structuring parameters. The musical characteristics each audification exhibited showed a strong causal relationship with each parameter tested. The effect of a nest's container sizes and the nature of its content (the band of random shorts in each tree and its width) was easy to observe and control. Overall, musical characteristics, such as rhythm, frequency, and timbre, were clearly affected by changes in these basic parameters. These changes can be traced to the binary representation of each nest and how structural aspects of the file changed with each parameter. Within the results sections for each parameter, I will discuss first how the values of the parameter affected the musical characteristics of each audification and then explain how JavaSerial's specifications and the byte content of the nests caused these characteristics to manifest.

4.1 EFFECT OF CONTAINER SIZE ON AUDIFICATION

Container size affected the frequency, rate of rhythmic oscillation, and length of each file's audification. Each of these characteristics can be linked explicitly to the size parameters of one or two of the nest's containers (Figure 15).

Samples in subgroup A acted as the control group for container size, where all container sizes were set to 50 iterations. The values for `l2_size` (the number of iterations of the outer list) were changed to 20 and 100 for subgroups B and C, respectively. The values for `l1_size` (the number of iterations of the internal list) were changed to 20 and 100 for subgroups D and E, respectively. The values for `t_size` (the number of iterations of the tree within the internal list) were changed to 20 and 100 for subgroups F and G, respectively. When comparing samples to determine the effect of container size on the audification, files where only the size changed (i.e. samples [00], [10], and [20]) were compared, as well as samples within a subgroup where a container's size remained constant (i.e. samples [10], [11], and [12]). The size of the outer list (`l2_size)` affected the audification the least. The only difference between samples where `l2_size` was changed (files in subgroups A, B and C) was the length of the audification (Figure 16).

Unsurprisingly, the larger `l2_size` was, the longer the resulting audification. Likewise, the shorter `l2_size`, the shorter the audification. The durations of samples within a group with the same container sizes was more or less consistent, with durations only varying by a second or so. This can be attributed to changes in the content of the nest. Durations across groups with the same `l2_size` varied widely, but this difference can be attributed to the change in value for other container sizes, which also affected the size of the test files (having only 20 iterations of a tree per internal list instead of 50 obviously results in a smaller file). The value of `l2_size` did not affect any other sonic qualities of the audification other than duration.

subgroup ID	sample ID	t1_size	l1_size	l2_size	range	rand_width	rhythm [y/n]	rate	pitch	timbre
A	[00]	50	50	50	low	narrow	y	medium	mid	broad
A	[01]	50	50	50	mid	narrow	n	--	[mid]	
A	[02]	50	50	50	high	narrow	y	medium	mid	nasal
A	[03]	50	50	50	low	mid	y	medium	mid	broad
A	[04]	50	50	50	mid	mid	n	--	[mid]	
A	[05]	50	50	50	high	mid	y	medium	mid	nasal
A	[06]	50	50	50	low	wide	y	medium	mid	broad
A	[07]	50	50	50	mid	wide	n	--	[mid]	
A	[08]	50	50	50	high	wide	y	medium	mid	nasal
B	[10]	50	50	20	low	narrow	y	medium	mid	broad
B	[11]	50	50	20	mid	narrow	n	--	[mid]	
B	[12]	50	50	20	high	narrow	y	medium	mid	nasal
B	[13]	50	50	20	low	mid	y	medium	mid	broad
B	[14]	50	50	20	mid	mid	n	--	[mid]	
B	[15]	50	50	20	high	mid	y	medium	mid	nasal
B	[16]	50	50	20	low	wide	y	medium	mid	broad
B	[17]	50	50	20	mid	wide	n	--	[mid]	
B	[18]	50	50	20	high	wide	y	medium	mid	nasal
C	[20]	50	50	100	low	narrow	y	medium	mid	broad
C	[21]	50	50	100	mid	narrow	n	--	[mid]	
C	[22]	50	50	100	high	narrow	y	medium	mid	nasal
C	[23]	50	50	100	low	mid	y	medium	mid	broad
C	[24]	50	50	100	mid	mid	n	--	[mid]	
C	[25]	50	50	100	high	mid	y	medium	mid	nasal
C	[26]	50	50	100	low	wide	y	medium	mid	broad
C	[27]	50	50	100	mid	wide	n	--	[mid]	
C	[28]	50	50	100	high	wide	y	medium	mid	nasal
D	[30]	50	20	50	low	narrow	y	fast	mid	broad
D	[31]	50	20	50	mid	narrow	n	--	[mid]	
D	[32]	50	20	50	high	narrow	y	fast	mid	nasal
D	[33]	50	20	50	low	mid	y	fast	mid	broad
D	[34]	50	20	50	mid	mid	n	--	[mid]	
D	[35]	50	20	50	high	mid	y	fast	mid	nasal
D	[36]	50	20	50	low	wide	y	fast	mid	broad
D	[37]	50	20	50	mid	wide	n	--	[mid]	
D	[38]	50	20	50	high	wide	y	fast	mid	nasal
E	[40]	50	100	50	low	narrow	y	slow	mid	broad
E	[41]	50	100	50	mid	narrow	n	--	[mid]	
E	[42]	50	100	50	high	narrow	y	slow	mid	nasal
E	[43]	50	100	50	low	mid	y	slow	mid	broad
E	[44]	50	100	50	mid	mid	n	--	[mid]	
E	[45]	50	100	50	high	mid	y	slow	mid	nasal
E	[46]	50	100	50	low	wide	y	slow	mid	broad
E	[47]	50	100	50	mid	wide	n	--	[mid]	
E	[48]	50	100	50	high	wide	y	slow	mid	nasal
F	[50]	20	50	50	low	narrow	y	fast	high	broad
F	[51]	20	50	50	mid	narrow	n	--	[high]	
F	[52]	20	50	50	high	narrow	y	fast	high	nasal
F	[53]	20	50	50	low	mid	y	fast	high	broad
F	[54]	20	50	50	mid	mid	n	--	[high]	
F	[55]	20	50	50	high	mid	y	fast	high	nasal
F	[56]	20	50	50	low	wide	y	fast	high	broad
F	[57]	20	50	50	mid	wide	n	--	[high]	
F	[58]	20	50	50	high	wide	y	fast	high	nasal
G	[60]	100	50	50	low	narrow	y	slow	low	broad
G	[61]	100	50	50	mid	narrow	n	--	[low]	
G	[62]	100	50	50	high	narrow	y	slow	low	nasal
G	[63]	100	50	50	low	mid	y	slow	low	broad
G	[64]	100	50	50	mid	mid	n	--	[low]	
G	[65]	100	50	50	high	mid	y	slow	low	nasal
G	[66]	100	50	50	low	wide	y	slow	low	broad
G	[67]	100	50	50	mid	wide	n	--	[low]	
G	[68]	100	50	50	high	wide	y	slow	low	nasal

[Group I]
[Group II]

Figure 14: Chart of audification test files

	Frequency	Rate	Length
Tree size:	x	x	x*
Inner list size:		x	x*
Outer list size			x

*implicitly affects: if a repeating structure changes in size and is repeated the same amount of times, the length of the audification will inherently change as well.

Figure 15: Effect of container sizes on musical parameters

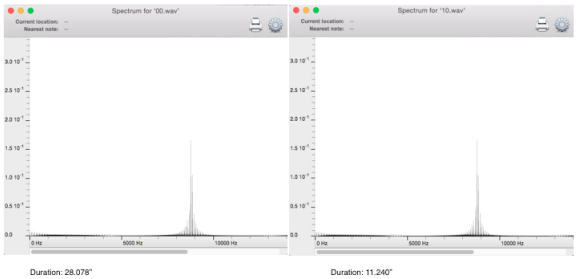

Figure 16: Spectrums and durations for samples [00] (left) and [10] (right)

The value of `l1_size` affected the rate of rhythmic oscillation in file Group I (the files that demonstrated a clear rhythm). Group II files (the files that manifested as noise) presented a special case resulting from the values of the random shorts inside the trees, and will be discussed later on. The value of `l1_size` did not affect Group II files (there was no sonic change between subgroups A, D, and E with Group II files). Within Group I, a smaller `l1_size` resulted in faster rhythmic oscillations, whereas a larger `l1_size` corresponded to slower oscillations. When comparing samples where only the `l1_size` parameter differed (i.e. between samples [00] and [30]), only the rate of oscillation changed (Figures 17, 18).

Changing the values of `t_size` affected both the underlying pitch and the rate of oscillation in each file's audification. When `t_size` was smaller, the underlying pitch was higher than the default parameter values and the rate of oscillation faster. Likewise, a larger value for `t_size` resulted in a lower pitch and a slower rate (Figures 19, 20).

Files within the same subgroup demonstrated the same pitch and the same rate of oscillation, and files where only the value for `t_size` changed exhibited the above changes in rate and frequency.

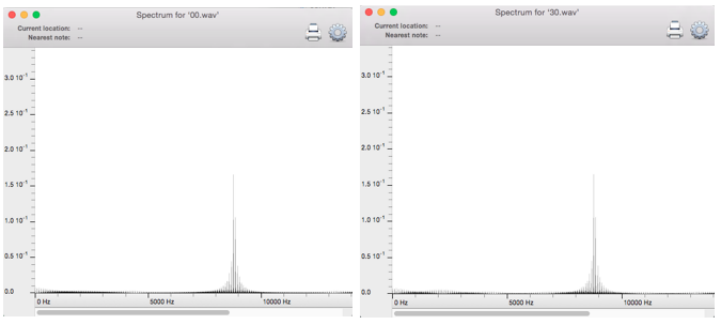

Figure 17: Spectrums for samples [00] (left) and [30] (right)

Figure 18: Waveforms for samples [00] (top) and [30] (bottom)

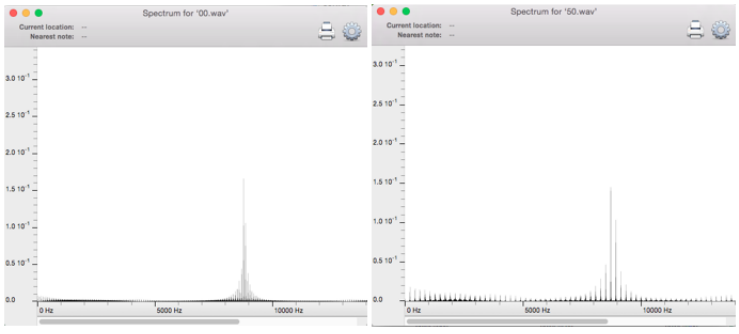

Figure 19: Spectrums for samples [00] (left) and [50] (right)

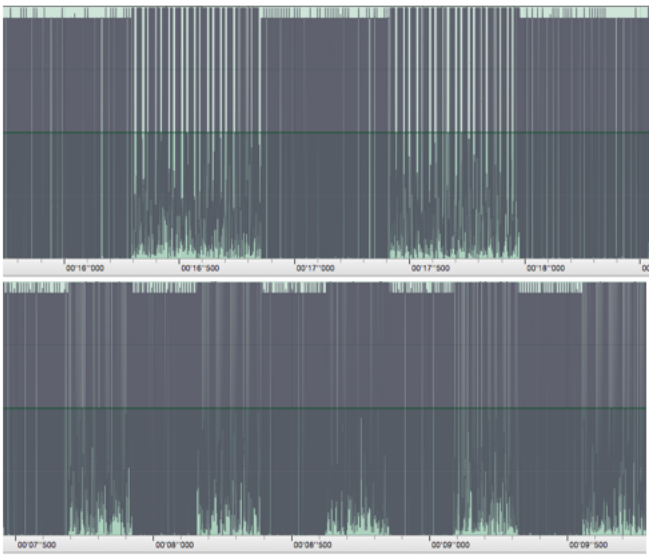

Figure 20: Waveforms for samples [00] (top) and [50] (bottom)

4.1.1 FILE STRUCTURE AND AUDIFICATION

When writing its output to disk, the JavaSerial library utilizes the following constants to denote terminal and constant values in a stream (Figure 20) [8].

With regards to container size, the value of interest from the above list is the `TC_ENDBLOCKDATA` [0x78] byte. This value appears at the end of each container iteration: it marks the end of each tree and each internal list instance. With the trees, this value has the same effect as the newline character in *.txt* files and the double word in *.bmp* files discussed in the introduction. It acts as a delineator, making the tree's byte representation a repeating sequence that manifests as tone in the audifications. Because each tree iteration is identical, this tone remains constant throughout each audification. For example, the length of one tree in sample [00] is 987 bytes. Using this value for the sequence length in the bit-length to frequency conversion equation discussed in the introduction produces a value of 89.36 Hz. This value corresponds with the fundamental frequency returned by FFT analysis on the audification (Figure 22).

The rhythmic oscillation present in the Group I files results from the `TC_ENDBLOCKDATA` byte marking the end of an internal list instance. This value is one byte, thus its appearance offsets the following data by a byte. Because the audio file utilizes a signed 16-bit bit-depth, a change in offset forces a change in each audio sample's value from iteration to iteration. Thus, one iteration of an internal list will manifest as tone, and the next as noise: shifting the offset by one byte makes many more zero-crossings happen in the audification's waveform, and results in a noisier sound [2]. Each iteration of an internal list is too long to manifest as a tone (as discussed in the introduction, the resulting

```
final static short STREAM_MAGIC = (short)0xaced;
final static short STREAM_VERSION = 5;
final static byte TC_NULL = (byte)0x70;
final static byte TC_REFERENCE = (byte)0x71;
final static byte TC_CLASSDESC = (byte)0x72;
final static byte TC_OBJECT = (byte)0x73;
final static byte TC_STRING = (byte)0x74;
final static byte TC_ARRAY = (byte)0x75;
final static byte TC_CLASS = (byte)0x76;
final static byte TC_BLOCKDATA = (byte)0x77;
final static byte TC_ENDBLOCKDATA = (byte)0x78;
final static byte TC_RESET = (byte)0x79;
final static byte TC_BLOCKDATALONG = (byte)0x7A;
final static byte TC_EXCEPTION = (byte)0x7B;
final static byte TC_LONGSTRING = (byte) 0x7C;
final static byte TC_PROXYCLASSDESC = (byte) 0x7D;
final static byte TC_ENUM = (byte) 0x7E;
final static  int   baseWireHandle = 0x7E0000;
```

Figure 21: List of constant and terminal symbols in the JavaSerial library

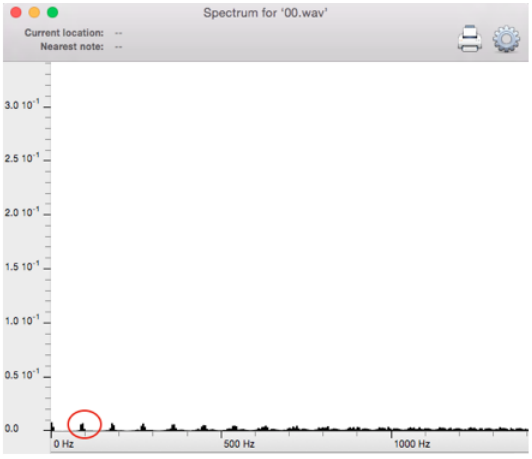

Figure 22: Spectrum for sample [00]

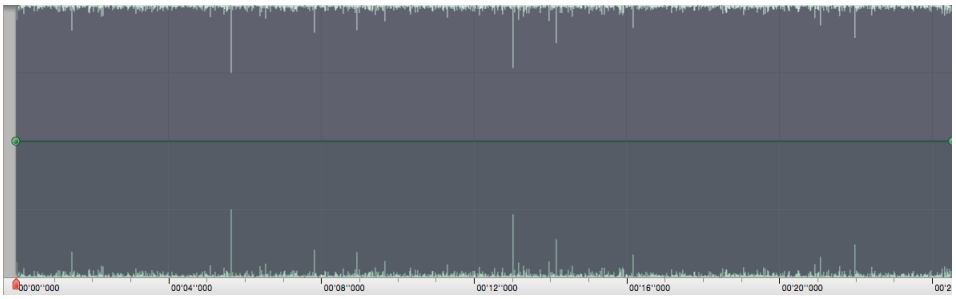

Figure 23: *Waveform for sample [01], a Group II file with no distinguishable rhythm*

frequency is too low), thus this repeating sequence appears as a rhythm in each audification. This is also why `t_size` affects the rate of oscillation in an audification: changing the size of the trees also changes the size of the internal list, thus changing the rhythm. Furthermore, changing the size of the internal list does not change the frequency of the tone present because the sequence is too large.

Overall, the value chosen for the container size parameter has a direct, controllable effect on the audifications produced by each file.

4.2 EFFECT OF RANGE ON AUDIFICATION

The three possible values for the range parameter of each file were `low` (all short values in the trees below zero), `mid` (values above and below zero), and `high` (all values above zero). This parameter's effect manifested as changes in timbre from file to file. It is also what caused the difference between files in Group I and Group II: whenever this parameter was set to `mid`, the audification presented as a noisy file, with no distinguishable rhythm (Figure 22).

This trend was consistent across file subgroups: within each subgroup, samples [x1], [x4], and [x7] (the files with the range parameter set to mid) were always Group II files.

The difference between files where range was set to `low` and range was set to `high` was slightly more nuanced. Within a subgroup, these files generated the same fundamental frequencies (had the same perceived pitch), but had slightly different timbres. Files with a `low` range parameter had stronger low frequency information, whereas files with a `high` range parameter had stronger high frequency information. This comparison was made between files in the same subgroup, where only the range parameter differed (Figure 23).

This correlation can be linked again to the binary representation of a nest. Each tree contained a series of random shorts. Because the tree representation acted as the repeating byte sequence that determined pitch, changes in its content without changes in its length results in a change in timbre: the waveform maintains the same period (and thus the same frequency), but its

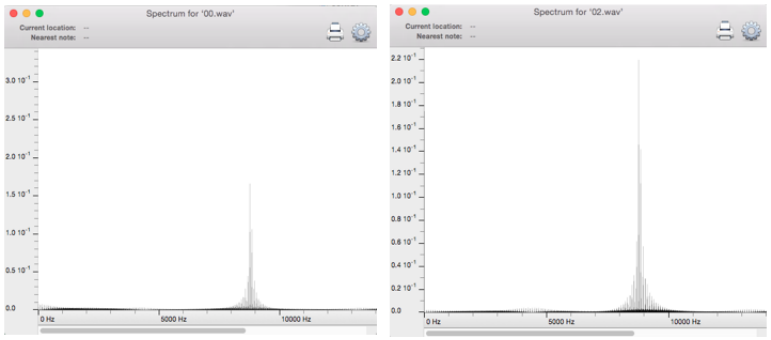

Figure 24: Spectrums for samples [00] (left) and [02] (right)

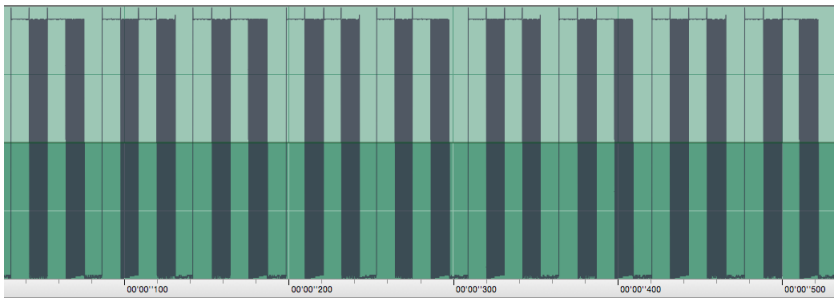

Figure 25: Portion of waveform for sample [00]

shape changes. The behavior of the timbral changes can be linked to the character of the random shorts. When the random range was set to `low`, each repeating sequence contained audio sample values above and below zero: the constant/terminal symbols used by JavaSerial provided the values above zero, and the random shorts provided values below (Figure 24). This created a balanced waveform with a clearly defined period, resulting in stronger amplitude values in the lower partials.

When the random range was set to `high`, all audio sample values were above zero (Figure 25). While the period of the wave was still clearly delineated (and thus the fundamental still perceptible), the unbalanced waveform resulted in higher amplitude values for the higher partials.

As previously discussed, when the random range was set to `mid`, each audification manifested as noise, without the clear rhythmic oscillation present in the other files. This is because the random shorts generated had values above and below zero: as a result, the higher magnitude bits in the binary representation of each short changed frequently. When converted into an audio waveform, this resulted in frequent zero-crossings, which obfuscated the repeating byte sequences created by the file's structure. Without these clearly delineated sequences, the rhythm and pitch present in the other files disappeared.

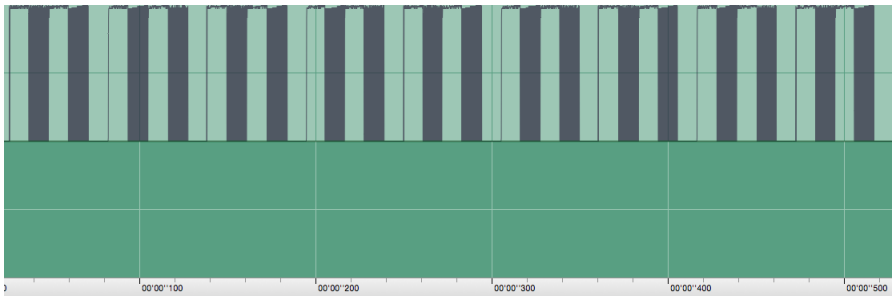

Figure 26: Portion of waveform for sample [02]

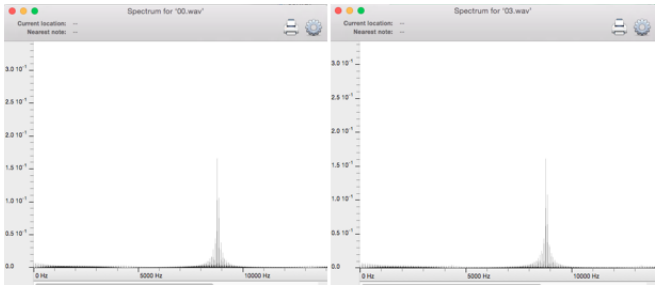

Figure 27: Spectrums for samples [00] (left) and [03] (right)

Again, changes in the random range parameter produced concrete, controllable changes in the musical characteristics of the audification. This is directly linked to the structure of the binary representation of each file.

4.3 EFFECT OF BANDWIDTH ON AUDIFICATION

The final parameter that I examined in this experiment was the width of the range of the random shorts (or *bandwidth*), with the possible values of narrow, mid, and wide. This parameter had the least effect on the audifications: changing it really only affected files in Group II. Changes of this value did not result in any change in Group I files, as confirmed by spectral analysis (Figure 26).

With Group II files, changes in bandwidth resulted in changes in the noisiness of a file. Files with a wider bandwidth had slightly more defined frequencies (Figure 27).

This, again, can be linked to the behavior of the random shorts. With a higher bandwidth (larger random range), the probability of a zero-crossing occuring decreases. Fewer zero-crossings result in less noise in the file.

While slight, changes in this parameter again resulted in clear changes in the audio.

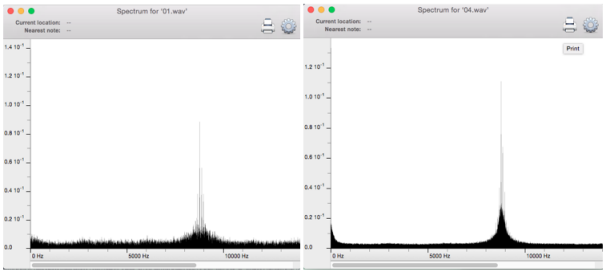

Figure 28: Spectrums for samples [01] (left) and [04] (right)

4.4 SUMMARY

Changes in container size resulted in changes in frequency, rate of rhythmic oscillation, and duration of audification. These changes can be linked to the binary representation of each file and the constants used by the JavaSerial library. Changes in the range of the random shorts stored inside each tree resulted in changes in the audification's timbre. When this parameter was set to mid, the file manifested as noise, with no rhythmic oscillation and a less-defined frequency. The bandwidth parameter only affected the Group II files: a wider bandwidth created less noisy files within this group. Overall, changes in each parameter resulted in clear, easily controllable changes in the audifications.

5 DISCUSSION AND CONCLUSION

As hypothesized, these results demonstrate that a generative process offers composers informed control over audification techniques. Each parameter tested caused explicit changes in the resulting audio file. However, this experiment tested a very small subset of variables in a highly controlled setting. Many more combinations and types of parameters remain to be tested, leaving an open field for future work on generative audification processes. Ideas for expansion in this area are as follows:

–Utilizing different data types within a nest: This experiment only tested nests containing random shorts. Future tests could easily examine the effects of utilizing other basic types, such as floats, integers, booleans or chars.

–Nests with different types of containers: This work examined a nest constructed from trees within a list within a list. These data structures could be swapped out for other types, such as hashmaps, to construct new types of nests. The binary representations of these data structures would be different, and would result in different types of audifications. However, I suspect that parameters such as container size would have similar effects on the audifications.

–Nests with more complexity: The nest used for this analysis had 3 layers. One could easily construct a nest with several more layers (5+) that could create

more musically complex audifications.

–Different programming languages: JavaSerial has a highly regulated specification for writing streams to disk, which is why the discussed files manifested in the way that they did. Other languages (C, Python) have ways of writing data structures to binary files that most likely use different specifications. Changes in specifications would result in changes in the file structure, thus producing different audifications.

While much work remains to be done with determining the scope of a generative audification process, this work demonstrates that such a process is viable in terms of giving composers concrete control over this technique. Furthermore, it proves that this method opens up a new arena in DSP research: manipulating binary structures to create sound can offer a wide array of sonic possibilities that can be controlled by composers. The sonic palette of this technique differs from all existing DSP techniques and is under-explored and researched. With any luck, this paper will motivate researchers to explore new methods of sound creation instead of redesigning existing algorithms that produce overused sounds.

6 Acknowledgments

This work was supervised by Roberto Hoyle as an honors thesis for the Oberlin Computer Science Department. I would also like to thank Thom Johansen (NoTAM) and Mikhail Malt (IRCAM) for their assistance with coding and implementation.

References

[1] Michael Scott Barren. Is the Image Hidden in Aphex Twin's "Equation" the Best Easter Egg in Electronic Music? *Vice*, 24(4), 2016.

[2] Paul Berg, Robert Rowe, and David Theriault. Ssp and sound description. *Computer Music Journal*, 4(1):25–35, 1980.

[3] Richard Boulanger and Victor Lazzarini. *The Audio Programming Book*. MIT Press, Cambridge, Massachusets, USA, first edition, 2011.

[4] Kim Cascone. The Aesthetics of Failure: "Post Digital" Tendencies in Contemporary Computer Music. *Computer Music Journal*, 24(4):12–18, 2000.

[5] Florian Dombois and Gerhard Eckel. *The Sonification Handbook*, chapter Audification, pages 301–324. Logos Publishing House, Berlin, Germany, 2011.

[6] Matt Donadio. How to Generate White Gaussian Noise. Available at http://dspguru.com/dsp/howtos/how-to-generate-white-gaussian-noise/http://dspguru.com/dsp/howtos/how-to-generate-white-gaussian-noise/, accessed March 13, 2017.

[7] Thom Holmes. *Electronic and Experimental Music: Technology, Music, and Culture*. Routledge, New York, 2016.

[8] Java Object Serialization Specification, accessed February 15, 2018. Available at https://docs.oracle.com/javase/7/docs/platform/serialization/spec/protocol.html.

[9] Thomas Knoll. Adobe Photoshop File Formats Specification. Available at https://www.adobe.com/devnet-apps/photoshop/fileformatashtml/#50577409_13084, accessed March 13, 2017.

[10] Look (Venetian Snares) Easter Egg - Look at Venetian's Cats, accessed September 30, 2017. Available at http://www.eeggs.com/items/46956.html.

[11] Curtis Roads. *The Computer Music Tutorial*. MIT press, 1996.

[12] Craig Stuart Sapp. WAVE PCM soundfile format. Available at http://soundfile.sapp.org/doc/WaveFormat/http://soundfile.sapp.org/doc/WaveFormat, accessed March 13, 2017.

[13] M. Serra. Stochastic Composition and Stochastic Timbre: GENDY3 by Iannis Xenakis. *Perspectives of New Music*, 31(1):236–57, 1993.

[14] Simplified Windows BMP Bitmap File Format Specification, accessed March 13, 2017. Available at http://www.dragonwins.com/domains/getteched/bmp/bmpfileformat.htm.

[15] Spectrograms of "3recurring", accessed September 30, 2017. Available at https://forum.watmm.com/topic/63401-spectrograms-of-3recurring/.

[16] ESynth DIY : Software for Generating ADSR Envelopes, accessed April 20, 2018. Available at http://hackmeopen.com/2011/12/synth-diy-software-for-generating-adsr-envelopes/.

[17] Barry Truax. Real-time granular synthesis with a digital signal processor. *Computer Music Journal*, 12(2):14–26, 1988.