UNIVERSITY OF CALIFORNIA, SAN DIEGO

AudioChalk: Idiosyncratic Software For Sketching Spectral Music

A Thesis submitted in partial satisfaction of the
requirements for the degree
Master of Arts

in

Music

by

Elliot M. Patros

Committee in charge:

Professor Tamara Smyth, Chair
Professor Tom Erbe
Professor Miller Puckette

2015

The Thesis of Elliot M. Patros is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California, San Diego

2015

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank my thesis committee members for their thoughtful advice, support, and imagination throughout my studies at UCSD. Tamara Smyth for introducing me to signal processing, patiently debugging with me, and for brainstorming with me during AudioChalk's development. Miller Puckette for sharing his experience in application development, celebrating small victories with me, and for bravely venturing from C into C++. Tom Erbe for actually teaching me how to code and for generally having incredibly practical advice.

I would also like to thank my friends and the UCSD community for sharing ideas, late nights in the Warren office, meals, dumb jokes, epiphanies, and everything else. Special thanks to Kevin Zhang for always knowing when to break for tossing a baseball or to get In-N-Out burgers. Chris Donahue and Kevin Haywood for technical support and tacos. Drew Allen for probably the most useful half-hour conversation I've ever had. Zachary Seldess for teaching me a bunch of really cool stuff. Caroline Miller for her creative influence and support.

Finally, thank you to my parents and to my brother Clay for their unconditional love and support, even from two thousand miles away.

ABSTRACT OF THE THESIS

AudioChalk: Idiosyncratic Software For Sketching Spectral Music

by

Elliot M. Patros

Master of Arts in Music

University of California, San Diego, 2015

Professor Tamara Smyth, Chair

This Thesis describes the design of the proof-of-concept software application, AudioChalk. AudioChalk, which combines a vector graphics editor with a oscillator bank synthesizer, lets users algorithmically generate, manipulate, or sketch partials by hand in the time frequency domain via intuitive graphical elements. The purpose of developing AudioChalk was to study, from a software developer's perspective, how content creation software for experimental musicians can be designed to offer both a flexible and efficient workflow. Simultaneously precise and efficient control are gained by sacrificing some of the software's flexibility. However, if losses of flexibility are carefully considered to not significantly interfere with intended use cases, users often do not perceive a loss of

possible functionality. The design choices regarding tradeoffs between efficiency and flexibility that were considered while developing AudioChalk are distilled into a rubric, called idiosyncrasy. Idiosyncrasy as a rubric attempts to generalize the consequences of software design choices from the intended users' perspective. The intended users in the case of AudioChalk are composers of experimental electronic music.

Chapter 1

Introduction

The job of the experimental composer, like that of any writer, is to communicate some idea to their audience. However unlike most other writers, the composer's ideas are communicated to their audience via a significant middle-man: their performer. It follows then that the composer's ability to effectively communicate with their performer directly affects their ability to effectively communicate with their audience.

The relationship between the composer and the performer is, regarding their communication, an *interpretive* relationship as the two can communicate with differing "degrees of explicitness".[1] As participants of an interpretive relationship, it is each member's responsibility to keep track of what the other knows about their intentions. For the composer, this means paying attention to how accurately their own ideas are received by the performer. For the performer, this means giving meaningful feedback to the composer about how well the composer's intentions are understood. Consider, for example, the typical workflow of a composer and a human performer. The composer starts by making a graphic score that explicitly communicates their musical intentions

[1] If *explicitness* is the opposite of *implicitness*, then the two terms describe either end of a spectrum that describes *interpretiveness*. According to linguistic philosopher Leo Hickey, "the more the hearer has to rely on contextual clues for the identification of the proposition expressed, the less explicit it is, and inversely" [Hic89].

to the performer. The performer then performs the score, which gives the composer explicit feedback about how well the composer's intentions were understood. Although this is a great start, in most cases, this explicit communication alone isn't enough to create a meaningful performance. For example, when a performer who's reading a score in rehearsal plays something the composer didn't intend, the composer doesn't always have to change the score to get a different result from the performer. (If this were the only way to communicate, things would get very frustrating very quickly.) Instead, the composer and performer can switch from explicit communication to implicit to increase the efficiency of their workflow; put simply, they can talk it out.

Implicit communication works wonderfully for humans, but how does it work for composers of electronic music? Electronic composers craft musical performances with specialized software applications instead of with human performers. Yet, partially due to the nature of musical software design, many electronic composers treat their software how they would treat simple tools rather than how they would treat a performer.[2] The large potential scope of music composition in general coupled with the deterministic relationship between electronic composers and their software means that many electronic composers have inefficient or restrictive workflows.

Unfortunately, it is difficult to have meaningful implicit communication with a software application since software is deterministic by definition. There is also no perfect solution for optimizing workflow efficiency for tasks with an undefined scope like, for example, experimental composition. Nevertheless, it is the software developer's responsibility to create efficient and open-ended software for experimental electronic musicians. It is therefore important for software developers to understand how tradeoffs

[2] A simple tool (e.g. a hammer) has a deterministic relationship with its user, as opposed to an interpretive relationship. Members of a deterministic relationship can only explicitly communicate, as efficiency isn't gained from the presence of contextual clues. (A hammer doesn't need to know why it's hammering something to work.) The efficiency of a deterministic relationship increases as the scope of the simple tool being used decreases and inversely [Ham02].

between efficiency and scope affect composers.

1.1 Objectives For Software Developers

Software developers, particularly those who intend to build helpful products for experimental electronic composers, have two important objectives. First, developers should continue extending the capabilities of software environments[3] in successful paradigms.[4] For example, developing compatible middleware, plugins, libraries, etc. for preexisting environments helps to improve the scope and efficiency of electronic composers' work. Second, developers should try to expand the number of useful paradigms. There are two ways to realize the latter objective. Developers can increase the number of useful paradigms by either improving on preexisting but poorly-designed paradigms, or by recombining components from potentially unrelated technologies into a useful new paradigm.

In any case, developers are tasked with optimizing both the efficiency and the scope of specific environments. Though the qualities of efficiency and scope are at odds with one another, optimizing for both tends to bring out another quality in software: *idiosyncrasy*. In other words, as software developers attempt to increase workflow efficiency and scope together, their software becomes more idiosyncratic. Before defining idiosyncrasy in more detail and explaining its potential use to developers, the current state of musical software should be discussed in more detail.

[3] Environment is defined here as a specialized software application for music composition.

[4] Paradigm is defined here as a style of environment that is categorized by the type of communication it support or encourages from composers.

1.2 Paradigms Of Experimental Electronic Composition

There are an enormous number of unique software environments for experimental music, and the vast majority of them belong to one of only three paradigms: Digital Audio Workstations or DAWs (e.g. Ableton, Pro Tools), patchers (e.g. Pd, Max/MSP), and specialized coding languages (e.g. SuperCollider, ChucK). To risk digression, one might expect for more than three well-established compositional paradigms to exist, especially in a medium for which process is so often an end in itself. The real issue though is how the lack of paradigmatic diversity partially contributes to the inefficiency of experimental workflows. Why? According to electronic composer Michael Hamman,

> "...software tools carry huge ideological and epistemological payloads that the human user must accept, silently or otherwise. Moreover, and as a consequence of their deterministic operation, those tools give no hint as to the enormous epistemic flexibility of which the computer is capable. Instead, humans are required to force problem formulations in ways that the computer can 'understand.' Under this rubric, *technique*—nominally defined as that through which work gets done—is reduced to the task of learning how to map one's notational view of a problem, or problem domain, into sequences of steps which have no meaningful relation to the relevant problem domain. Such environments produce little in the way of pleasure—the tool, rather than being a 'liberator' of human beings, becomes an agent of repression, forcing the user to succumb to a normalizing view of her/his task environment." [Ham02]

One could generate a figuratively endless list of significant factors that contribute to the lack of paradigmatic diversity. This is an important discussion to have, though it is beyond the scope of this thesis. To get the ball rolling just a little: there are financial reasons, for example feedback loops created by market demand for well-established paradigms that pressure developers to prioritize working on clichéd environments; there are technological reasons, for example how networks of cross-platform dependencies and communication standards are in place for well-established paradigms but that are

prohibitively difficult for small dev teams to build from the ground up; educational reasons, for example how many software users (e.g. composers) often lack the knowledge required to build new, though much needed, tools for themselves; or proprietary reasons, for example the lack of open sourced code that would allow users to customize existing environments to better suit their needs; to name a few. Again, a further discussion of these and other factors that contribute to the homogeneity of compositional software are all beyond the scope of this document.

Thankfully, some developers are in a position to expand on paradigms of experimental compositional software. Yet though the language needed for them to describe interactive software design complexities exists [MF97], is not all that common. The objectives of software designers will hopefully be made both more clear and obtainable by presenting design tradeoffs as a rubric in the discussion on idiosyncrasy.

Now I will introduce a proof-of-concept software environment, called *AudioChalk*, that I developed in parallel to the software-design rubric idiosyncrasy. In and of itself, AudioChalk is *not* intended to be a solution to any of the problems introduced above. Rather it is only one interpretation of how content creation software can be intentionally idiosyncratic, and was developed to study how idiosyncrasy can benefit users.

Chapter 2

AudioChalk

AudioChalk is a proof-of-concept musical drawing software application that combines a vector graphics editor with an oscillator bank synthesizer.[1] It allows users to synthesize sounds, which can range from abstract to realistic, by drawing and manipulating graphical lines and curves, called *tracks*, with a mouse and keyboard. Tracks, which are drawn on a *canvas* that represents the time-frequency domain,[2] represent spectral partials and allow users to control the pitch and loudness of sinusoidal oscillators over time.

The target user of AudioChalk is the experimental electronic composer. More specifically, AudioChalk is intended for experimental electronic composers who are interested in simultaneously controlling both small-scale details and large-scale formal structures of their music with a single tool. Therefore, AudioChalk can be used as a sound editor, as a workstation for experimental electronic composition, or as both at once.

As a sound editor, AudioChalk improves both the ease of use and the fidelity

[1] For now, AudioChalk is only compatible with Mac OSX.

[2] AudioChalk's canvas represents time moving forward from the left of the screen to the right, and pitch ascending from the bottom of the screen to the top.

of some standard DSP techniques, for example independent time stretch and frequency shift, over many time domain editors by nature of working in the frequency domain. As a compositional workstation however, its use of the frequency domain is more problematic. Since a single sound can contain thousands of partials or more, a typical AudioChalk session often represents an overwhelming amount of data, even for experienced composers. The concept that AudioChalk intends to prove is given purpose with this problem: how can a software application allow composers to effectively control huge amounts of musical data without sacrificing either their ability to access low-level parameters or limiting the scope of their musical projects? Put another way, if there's a tradeoff between the efficiency and scope of a workflow, how is a user's experience affected by optimizing for one, the other, or for both?

The specific solution that AudioChalk uses to address the problem of workflow tradeoff between scope and efficiency is presented below in a more detailed overview of its features. Later, the problem is generalized for other paradigms of musical software in a overview of *idiosyncrasy* as a rubric for developers who are concerned about optimizing user-experience.

2.1 The Main Window

The AudioChalk application uses only one main window that contains its entire graphical user interface, or *GUI*. The GUI is split into four components, all of which simultaneously accept user input and display information about the state of the AudioChalk session. Figure 2.1 shows the initial state of AudioChalk's main window when a new session is opened. The four components of the main window are:

A. Buttons for common actions, which are greyed out when either inactive or disabled. From left to right the buttons are on the top row: play/pause and stop buttons for in-application audio playback, audio preferences window button, and zoom

in button. From left to right on the bottom row: line drawing tool button, pencil drawing tool button, selector tool button, and zoom out button.

B. On the top is the track loudness automation window, which lets users define loudness curves for new tracks. Below that is the play progress slider, which lets users set and see the in-application playback position.

C. The integrated preferences window. The pull down menu above allows users to swap between preferences windows to improve, depending on their task.

D. The canvas allows users to draw on and displays tracks. The canvas has the following musical properties. It has a minimum and maximum frequency, which are located at the bottom and top of the canvas respectively. It also has a start and end time, which are located at the left and right of the canvas respectively.

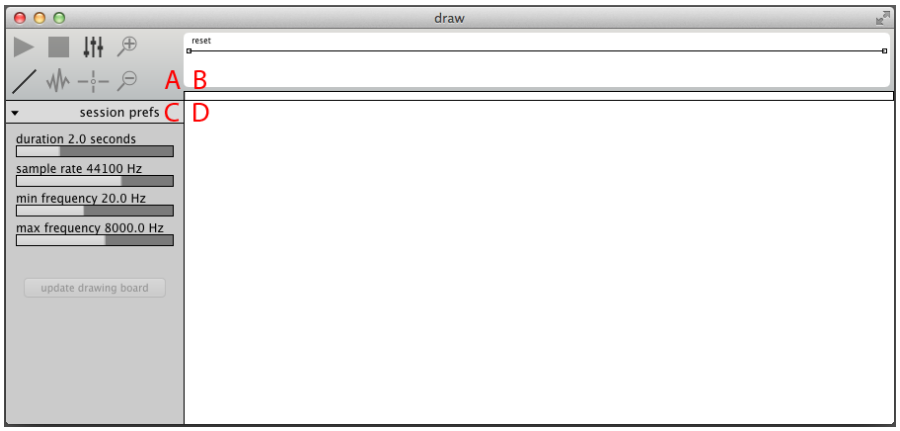

Figure 2.1: AudioChalk opened before any user input.

In addition to the main window, users can also interact with a Mac native array of pulldown menus at the top of the screen, which have a button for every command that AudioChalk allows except for drawing commands. Finally, AudioChalk has an extensive keyboard shortcut mapping to increase efficiency for experienced users. Keyboard shortcuts and menu navigation guides are mentioned alongside their respective features throughout this Chapter.

The primary focus of AudioChalk is to make sound by generating tracks on the canvas. Users can generate tracks in one of two ways, by drawing them with a mouse, and by importing audio with the import audio feature.

2.2 Generating Tracks With Drawing Tools

AudioChalk offers a minimal but effective array of tools for drawing tracks on the canvas. With either the line tool or track tool, users can *click and drag*[3] on the canvas to draw new tracks. The line tool creates tracks with only a start and end point, and are connected by a straight line. The track tool works just like the line tool, except that it can create tracks with additional intermediate points.

Tracks are both graphical and musical objects. The relationship between their graphical and musical elements is simple and well-defined. Though they are intended to be intuited by users who are familiar with traditional time-frequency domain displays, their behavior deserves a brief introduction.

2.2.1 Anatomy Of A Track

Tracks are made of 2 to *n* spectral peaks, called simply *peaks* for short. A peak is a three dimensional point, or coordinate. In the audio domain, a peak has the properties *time* in seconds, *frequency* in Hz., and *loudness* in dB. In the graphical domain, time runs along the *x* axis, frequency along the *y* axis, and gain along the *z* axis, which is unseen to the user. So, the position of a peak on screen is defined by their sonic properties with respect to the sonic properties of the canvas. Gain is described in more detail in Section 2.2.4, *Drawing And Seeing Loudness*. Peaks are not seen by users directly. Rather, their position is inferred by *track segments*, which are made by connecting two adjacent peaks. So, two adjacent peaks define either side of a *track segment*.

A track segment is a linear interpolation of the parameters of both its peaks, both graphically and sonically. In other words, a track segment is a line segment graphically, and a linear sinusoidal chirp sonically. The track segment is the most basic element of

[3]Click and drag is a three part gesture that contains a mouse down event, a mouse drag event, and a mouse up event. Each event has a corresponding X-Y pixel coordinate.

AudioChalk that can be seen or heard by a user. Track segments are building blocks of *tracks*.

A track is generated by concatenating 1 to *n* track segments end to end, where its oscillator has continuous phase. Tracks with one track segments can be drawn with either the line tool or the pencil tool, which are described below. Tracks with multiple track segments can only be drawn with the pencil tool.

Finally a track group, which will be introduced in more detail in Section 2.4.2, *Track Groups*, is a collection of 1 to *n* tracks, for which audio is pre-rendered and the graphical object is frozen from any potential editing until it is *ungrouped*.

For now it is enough to know that AudioChalk sessions contain hierarchical data structures that are building blocks of each other; all of which are available for users to interact with in compositionally meaningful ways.[4]

2.2.2 The Line Tool

The line tool creates two-peaked tracks, which create linear sinusoidal chirps when rendered to audio. Because perfect linear chirps don't happen in nature, the line tool is better at building stereotypical electronic music sounds than realistic sounds.

The first peak of any new line track is located at the time-frequency coordinate that corresponds to the X-Y pixel coordinate of the mouse down event that generated it. Similarly, the last peak of any new track is located at the time-frequency coordinate that corresponds to the X-Y coordinate of the following mouse up event. If the coordinates of the mouse down and mouse up events are the same, a new track isn't generated. During mouse drag events, a semi-transparent track is drawn on the canvas as a preview.

[4] AudioChalk's compositional building blocks are, in order from lowest to highest level:
Peak → *Track Segment* → *Track* → *Track Group*.

2.2.3 The Pencil Tool

The pencil tool works the same as the line tool but with the added possibility of concatenating multiple track segments. This means the pencil tool is able to create more complicated frequency trajectories over time than the line tool.

Similarly to the line tool, the first and last peak of the pencil tool are located at the time-frequency coordinates that correspond to the X-Y pixel coordinates of their respective mouse events. Unlike the line tool, the pencil tool responds to a user-specified parameter, *interpeak distance*,[5] which specifies the minimum distance between peaks in milliseconds.

While dragging a new track with the pencil tool, the position of mouse drag events are compared to the position of the last valid peak. When the distance between a mouse drag event and the previous valid peak is greater than or equal to the interpeak distance, a new peak is appended to the track. This means that the pencil tool is not able to draw both backward and forward in time, though this makes sense because time is much less flexible than that in real life anyway. Also similarly to the line tool, the pencil tool creates a semi-transparent preview of the new track during mouse drag events.

2.2.4 Drawing And Seeing Loudness

Using the line and pencil tools to draw tracks lets users shape the frequency trajectories of oscillators over time. A description of how the third dimension of oscillator bank synthesizers, *loudness*, is drawn and represented has been ignored so far.

[5] A slider for users to define *inter-peak distance* is located in the *track editor* window of the integrated preferences window.

Seeing Loudness

Regarding the graphical representation of loudness, other frequency domain audio applications typically map specific colors to specific degrees of loudness. SPEAR [Kli09] for example, uses a two color scheme for representing loudness: the loudest parts of tracks are opaque black, and the quietest parts of tracks are transparent black (see-through), which blend into the white canvas. Loudness gradients are graphically represented as a transparency gradient in this scheme. This is an intuitively user-friendly solution in that quieter spectral peaks are both less visible and less audible, and inversely. In other words, loudness is visually perceived by users as a loudness *topology* with a two color scheme, which lets users quickly infer the relative importance of all spectral peaks in view.

However, a two color scheme has two important drawbacks. First and most obviously, very quiet partials are difficult to see because they blend into the canvas. This increases the risk of users accidentally overlooking very quiet partials when editing large numbers of tracks, and leads to mistakes that are often noticed too late to easily fix.

The problem of quiet partials blending into the background is solved in other applications by using a high-contrast three color scheme. AudioSculpt for example [IRC10], uses a high-contrast three color scheme that increases the visibility of all partials, including very quiet partials. In a three color scheme, the canvas is black, and loudness corresponds to a high-contrast *hue* gradient where, for example, loud is orange and quiet is blue. Though this scheme improves the visibility of quiet partials, it, like the two color scheme, has another important drawback: the relationship between relative loudnesses and relative transparencies of colors is fixed.

The fixed relationship between color and loudness is problematic because it cannot be perceptually equivalent for all users in all contexts; some users will inevitably infer at some point that a partial is significantly quieter or louder than it appears to be.

The perceptual discrepancy is both important and difficult to unlearn for two reasons. First, context affects one's perception of relative loudness differently than it affects one's perception of visual contrast. For example, if all partials on screen have a narrow range of loudness, their relative loudnesses still strongly influence the timbral quality of the final sound. However, the slight differences between the colors representing those partials will undermine the importance of their relative loudnesses. Second, many people will rely more heavily on a visual representation over an aural representation of sound when given a choice between the two. In other words, the visual suggestion of loudness significantly skews one's aural perception of loudness, which leads to creating sounds that are different than the user intends.

The drawbacks mentioned above suggest that an ever-present visual representation of loudness can actually decrease the efficiency of frequency domain workflows, despite its other advantages. It follows that, at least in some cases, a less intrusive visual representation of loudness would increase a user's tendency to rely on their ears rather than on their eyes to determine loudness, and ultimately increase the efficiency of their workflow.

Drawing Loudness

AudioChalk uses the same method to author loudness as it does to visually represent loudness. When tracks are drawn with either the line tool or the pencil tool, that track inherits the loudness curve displayed on the track loudness automation window (see Figure 2.1, part B). The loudness of a track in AudioChalk can be determined in three ways. First, and as suggested above, by listening to it, though over-auditioning sounds can be time consuming and eventually cause ear fatigue. The second method of determining loudness is to select a track with the selector tool (the selector tool is discussed more in Section 2.4.1, *Track Editing Tools*), which causes the track loudness automation window

to display the automation curve for the selected track. Selecting a track to view its loudness curve also allows users to edit a track's loudness. Finally, multiple tracks can be quickly made into track groups, which changes the visual representation of those tracks into a time-domain waveform that inherently conveys loudness. Track groups are useful for a number of reasons, and are discussed in more detail in Section 2.4.2, *Track Groups*.

2.3 Generating Tracks By Importing Audio

Drawing tools are not the only way for users to generate tracks in AudioChalk. Users can also import audio files, which causes AudioChalk to automatically draw tracks that will reconstruct that sound. The algorithm that AudioChalk uses for making tracks from audio files will be described in more detail in Section 2.3.1), *Getting Tracks From Audio Files*. The ways in which users can interact with AudioChalk's audio import feature will also be described in more detail in Section 2.3.2, *Rapidly Auditioning Imported Audio*.

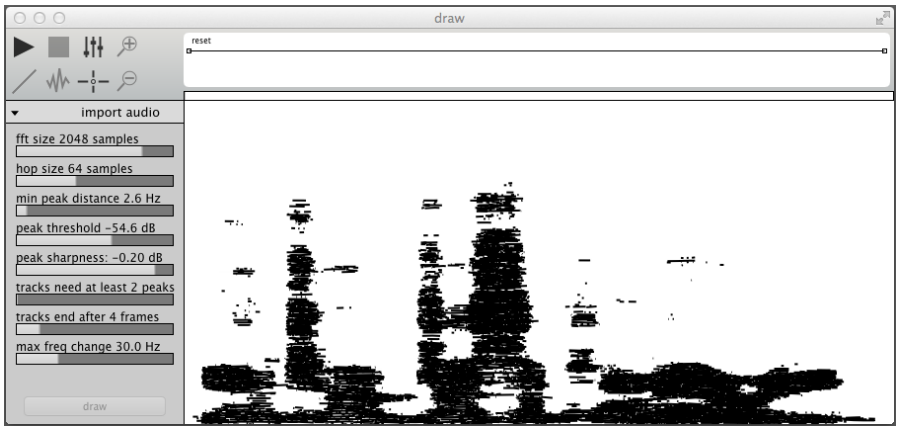

Figure 2.2: Audio from speech imported into AudioChalk.

Details aside, the ability to import audio files into AudioChalk's session format provides users with two useful tools. First, as mentioned at the beginning of this Chapter, some sounds contain thousands of partials, each of which may contain hundreds or more

spectral peaks with unique *time-frequency-loudness* coordinates. In the case where a user intends to generate spectrally dense sounds, the audio import feature can reduce the number of clicks required by orders of magnitude. The rapid track generation of the audio import feature will also free up time for users to play with AudioChalk's track editing features, which are discussed in more detail in Section 2.4, *Track Editing Tools*. Second, many interesting sounds do not have intuitively obvious spectral envelopes, or topologies. AudioChalk's audio import feature encourages users to discover interesting spectral structures of complex sounds.

2.3.1 Getting Tracks From Audio Files

The process of getting tracks from audio files in AudioChalk happens in three steps. First, the waveform of the user-selected audio file is cooked and its samples are stored in an array as floating point numbers. Second, the DFT of each frame is calculated in order to find the most significant spectral components of the imported sound. The significant spectral components of the audio file, called *peaks*, are stored in a *peak matrix*, which are used to draw tracks in the final step. Finally, tracks are built by tracing, or in other words connecting the dots, through available peaks in the peak matrix. These three processes are explained in more detail below.

Reading And Cooking Audio Files

Specialized *JUCE* [Ltd14] classes are used in AudioChalk to read audio file formats and to copy their contents into time domain buffers. This makes many standard uncompressed audio file formats compatible with AudioChalk (e.g. .wav, .aif). After the audio file is read into a buffer, its samples are cooked as follows:

1. Multichannel audio files are summed to mono, because AudioChalk only supports mono audio for now.

2. The audio is resampled to match AudioChalk's session sample rate, if necessary.
3. In preparation for peak picking, both the beginning and the end of the audio buffer are given zero padding.
   A. A first zero pad, with a size in samples equal to one half the size of the FFT window, is prepended to the audio buffer. This is to add sharpness to the attack of sounds reconstructed from spectral signals [Zï1].
   B. A second zero pad, with a size in samples equal to the amount required to bring the total duration of the buffer to the next largest multiple of the FFT window size plus an empty frame, is appended to the audio buffer. The size of the second zero pad, p, is expressed as:

$$p = (hop - dur \bmod win) + hop \tag{2.1}$$

   Where *hop* is FFT hop size in samples, *dur* is duration of buffer in samples after step 3.A., and *win* is the FFT window size in samples.
4. The audio buffer is normalized for the benefit of the peak picking algorithm. However, the gain coefficient from this step is saved in order to repair the buffered audio to its original amplitude before it is drawn.

Spectral Peak Picking

The purpose of peak picking is to extract only the significant spectral components from each frame of the imported audio. Because the process relies on DFT, standard DFT-related side effects must be accounted for. The limitations and common artifacts of the DFT are well known, and a detailed discussion of them are beyond the scope of this thesis. To read more on the DFT, particularly as it applies to peak picking, I recommend starting with Xavier Serra and Julius O. Smith III's publications, particularly their work on Spectral Modeling Synthesis [SJS90].

Being a tool for composers, the peak picking algorithm used in AudioChalk is intended to err on the side of "sounding good" over "being accurate". Though the algorithm can produce significantly flexible results depending on the intentions of the user and the import settings they use. The algorithm works as follows:

For each frame:

1. The DFT is calculated.[6] All values but the real values of all bins between (though not including) the DC and Nyquist frequency bins are filtered out. The remaining values are *peak candidates*.

2. The gains of peak candidates are converted to dB. and normalized. The minimum magnitude is -256 dB. Magnitudes below -256 dB. are clipped. This value was chosen to prevent unnecessary comparisons between bins that are significantly quieter than quantization error with 24 bit audio. The value -256 could probably be raised substantially without creating perceptible artifacts. Converting gain to dB. magnitude improves the accuracy of parabolic interpolation used later to increase the precision of bin frequency and amplitude estimates [SJS90].

3. Peak candidates are filtered for loudness. Bins quieter than the user specified minimum[7] are filtered out.

4. Peak candidates are filtered for sharpness.[8] Bins that are not at least as sharp as the user specified minimum[9] are filtered out.

5. Peak candidates are sorted from loudest to quietest to prepare for the next and final step. Since the computational complexity of sorting depends on the size of the array to sort, this step is reserved for last when the size of the array is as small as possible.

6. Finally, peak candidates are filtered for minimum peak distance.[10] Any bins that are not at least the user specified minimum peak distance of all louder bins in their frame are filtered out. All remaining bins in this frame are considered valid peaks.

7. At this point, each peak's position in frequency and loudness is refined by parabolic interpolation as mentioned in step 1.

8. Finally, the list of valid peaks in this frame is appended to the peak matrix, from which tracks are rendered in the next step, *partial tracing*.

---

[6] The DFT is calculated with the fftw 3.3.4 library [FJ14].

[7] see Table 2.1, *Minimum peak loudness*.

[8] Sharpness is the degree of loudness a frequency bin has relative to its immediate neighbors. For example, three adjacent bins of magnitudes (in dB.) $[-7,-1,-5]$ have a sharper peak than three adjacent bins of magnitudes $[-1,0,-2]$. The previous are examples of peaks with convex sharpness; concave sets are not peaks and do not have sharpness. Furthermore, sharpness is defined by the amount the peak is louder than its nearest neighbor. Therefore considering our previous examples again, the former has a sharpness of 4 dB., and the latter has a sharpness of 1 dB.

[9] see Table 2.1, *Peak sharpness*.

[10] see Table 2.1, *Maximum frequency between adjacent peaks*.

For peak picking, frames advance by the user specified FFT hop size.[11] Interestingly, the FFT hop size is *not* necessarily a function of FFT window size in AudioChalk. Since AudioChalk tracks are not rendered with iFFT synthesis, there is no gain penalty for using very small hop sizes. Users can therefore independently customize FFT hop sizes and FFT window sizes to suit the aural qualities imported sound with minimal artifacts.

Partial Tracing

The array of peak lists for each frame creates a two-dimensional *peak matrix*, which is used to trace partials that form tracks. The peak matrix is structured so that frames of peaks become the matrix' columns. The peaks in each column are sorted by frequency. As a side not, the term matrix here is used loosely, because not all columns are guaranteed to have the same number of peaks. Though this data structure is more accurately called a second order list, the term matrix is a much nicer word to read.

The partial tracing algorithm uses two arrays of tracks: a temporary array for tracks that are "in progress" (called *search tracks*), and a permanent array for tracks that are "valid" (called *valid tracks*). At the end of the partial tracing algorithm, the search tracks array and its contents are deleted. The peak matrix and its contents are also deleted, and any unused peaks and tracks remain unused. The valid tracks array is the array from which tracks are finally drawn. This happens by appending the contents of the valid tracks array to the track array from which AudioChalk normally draws.

AudioChalk uses the following algorithm for tracing partials through the peak matrix:

[11] see Table 2.1, *FFT hop size*.

For each frame:

> For each peak, append search tracks with peaks from the current frame, or start a new search track:
>
> > If a search track is within the user specified minimum peak distance[12] and does not already have a peak in it from this frame:
> >
> > > Add this peak to the nearest available[13] search track by frequency.
> >
> > Else:
> >
> > > Create a new search track with this peak.
>
> Clean up search tracks. . . for each search track:
>
> > If a track hasn't found a new peak in the user specified amount of time,[14] it is called an *old track*:
> >
> > > If the old track is too small:[15]
> > >
> > > > Remove and delete the old track from search tracks.
> > >
> > > Else:
> > >
> > > > Remove the old track from search tracks and append it to valid tracks.

By now, any tracks left in search tracks are invalid and are deleted. The tracks that have been added to the valid tracks array represent the imported sound as specified by the peak and track finding parameters.

2.3.2 Rapidly Auditioning Imported Audio

The process of importing audio, picking peaks, and tracing partials has no optimal solution for all types of sounds that users may import, nor is there an optimal solution for all possible musical intentions. To increase the flexibility and efficiency of the audio import feature, users are given access to a number of parameters that affect peak picking and partial tracing, as well as the ability to quickly audition and reimport audio. The parameters for importing audio are presented in Table 2.1.

[12] see Table 2.1, *Minimum peak distance*.
[13] Available search tracks are those that do not already have a peak from the current frame.
[14] see Table 2.1, *Maximum time between adjacent peaks*.
[15] see Table 2.1, *Minimum peaks per track*.

Table 2.1: Peak and track finding parameters that are available to users via the integrated preferences windows shown in Figure 2.1, part C.

Musical Parameters	**Unit Of Measurement**
FFT window size	samples
FFT hop size	samples
Minimum peak distance	Hz.
Minimum peak loudness	dB.
Peak sharpness	dB.
Minimum peaks per track	integers
Maximum time between adjacent peaks	frames
Maximum frequency between adjacent peaks	Hz.

The peak and track finding parameters are available to users as sliders as shown in Figure 2.4. The bottom of the import audio preferences window has another important component: the *draw button*, which reimports and redraws imported audio with updated parameter settings. The import audio preferences window as a whole can be used to greatly increase the flexibility, efficiency, and intuitiveness of importing audio, as is explained below.

Audio is initially imported in one of two ways. First, by selecting the **File** >> **Import Audio...** path from the native pulldown menu. Alternatively, by using the keyboard shortcut **cmd** + **i**, and selecting an audio file from the file explorer popup window. The initial audio import follows the parameters set by users with the import audio preferences window. Given the nature of track finding, it is expected that users will have to redraw the results a few times to get it right. This is where the draw button at the bottom of the import audio preferences window becomes handy. AudioChalk saves the cooked waveform of imported audio internally, so that users can adjust audio import settings and redraw imported audio without having to repeatedly navigate the file explorer popup window, and without having to repeatedly reenter complicated or unintuitive parameters. Rather, users are encouraged to dial in parameters with a system optimized for trial and error.

The import audio preferences window together with AudioChalk's in-application audio playback, lets users quickly audition import parameters and fine-tune settings for a particular type of sound and musical intention.

2.4 Track Editing Tools

So far, AudioChalk's tools for *generating* tracks have been described. AudioChalk also contains a toolset for *editing* or manipulating tracks that have already been generated. The track editing tools built into AudioChalk are intended to give users the ability to accomplish a variety of DSP techniques and other audio editing tasks.

2.4.1 The Selector Tool

The selector tool is the most basic tool of AudioChalk's track editing feature set. The selector tool is used for selecting tracks that the user intends to manipulate with one of the other track editing features.

A user can select a *single* track by clicking on it (or near it with a small error tolerance) with the selector tool. If the mouse hits a track, all but the selected track are dimmed slightly, and the selected track becomes outlined by a bounding box to indicate which track was selected.[16] A selected track can be deselected in one of three ways. First, by choosing **Edit** >> **Deselect** from the native pulldown menu. Second, by pressing the keyboard shortcut **Esc.**. Finally, by clicking outside of the bounding box without hitting another track.

A user can select *multiple* tracks by clicking and dragging to trace a rectangular selection region with the mouse. This action selects all tracks that intersect with the selection region. Multiple selected tracks are both displayed and deselected in the same

[16] see Figure 2.3 for an example.

way that a single track is displayed and deselected.

Other useful select tool tricks include *select all*, *deselect all*, *select inverse*, and *non-continuous selection*.

The *Select all* command selects all tracks in the session, and can be called by the native pulldown menu **Edit** >> **Select All**, or by the keyboard shortcut **cmd** + **a**.

The *deselect all* command deselects all selected tracks if there are any. It can be called with the native pulldown menu **Edit** >> **Deselect All**, or with the keyboard shortcut **Esc**.

The *select inverse* command inverts the selection state of all tracks, so that selected tracks are deselected and unselected tracks are selected. It can be called with the native pulldown menu **Edit** >> **Select Inverse**, or with the keyboard shortcut **cmd** + **shift** + **i**.

Non-continuous selection is a passive feature that lets users click on individual tracks, or click and drag over multiple tracks, to add them to an existing selection group. This is accomplished by clicking on individual tracks, or over multiple tracks, while holding **cmd**. Tracks that are selected can be non-continuously deselected the same way.

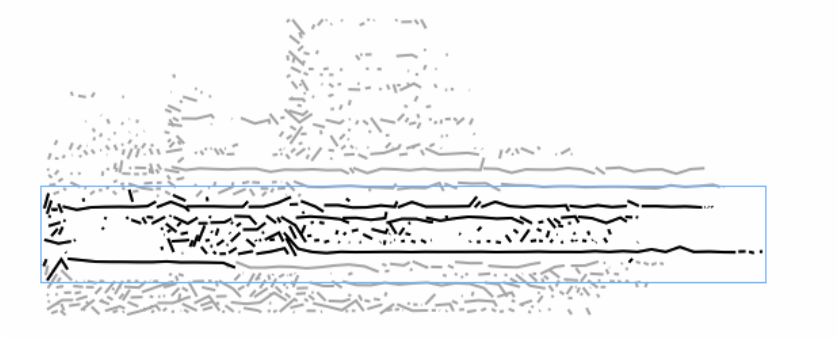

Figure 2.3: Multiple tracks from a piano chord progression are selected and highlighted by a bounding box. Unselected tracks are dimmed.

2.4.2 Track Groups

One of the issues users will face when using AudioChalk is a difficulty in both identifying and in maintaining the individuality of partials belonging to different sounds.

For example, if more than one sound are imported and overlapped in either time or frequency, how should users keep their tracks separate? This is what *track groups* are for.

Track groups are made from one or more selected tracks. To make a track group *freezes* those tracks, or in other words, makes those tracks *ineditable*. The make track group command is called by using the selector tool to select tracks, and then pressing the keyboard shortcut, **cmd** + **g**. In addition to being frozen, the audio for grouped tracks is rendered and assigned to the track group. This can be useful for speeding up the process of auditioning and bouncing audio when sessions have many tracks.

Track groups are painted differently than other tracks to distinguish them from the non-grouped tracks. When a new track group is created, first, its *z-order* is set to 0. In other words, it is sent to the farthest visual layer away from the users perspective. Next, the bounding box is used to bound a semi-transparent time-domain waveform display of the tracks in the group. The waveform display is semi-transparent so that users can see if any track groups are stacked on top of each other.

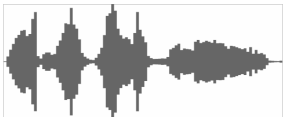

Figure 2.4: A track group made from imported speech.

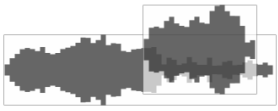

Figure 2.5: Multiple track groups are semi-transparent to convey depth and to prevent them from visually obscuring each other.

Track groups, though frozen, are not permanently frozen. Users can select track groups by either right clicking on them (or by pressing **ctrl** + **click** with a mac that uses a trackpad instead of a mouse), or by holding **cmd** while right clicking to cycle through the selection of all visible track groups. The latter method is useful when multiple track

groups are stacked vertically. For now, track groups cannot be nested mostly because I'm not sure if that's a useful feature.

The existence of track groups suggests a technique for users who intend to compose with multiple distinct sounds in one AudioChalk session. If track groups are created for unique sounds[17] as they are generated, the user can *ungroup* the sound they want to edit and *regroup* it immediately when they're done. Put another way, all sounds in AudioChalk, other than the sound being edited currently, should be in track groups. This prevents any stray partials from being accidentally either excluded or included in mass track edits. Treating track groups this way may also be used effectively with select tool shortcuts like *select all*, which have no affect on track groups.

AudioChalk's various track editing features and DSP effects will now be described in more detail.

2.4.3 Linear Transform

Selected tracks can be linearly transformed across the time domain, frequency domain, or both at once. A linear transform is performed by clicking and dragging within the bounding box of selected tracks. Alternatively, single tracks can be simultaneously selected and linearly transformed by clicking on them and dragging. A linear transform may be locked to a single dimension, either time or frequency, by holding **shift** while dragging, which lets users force tracks to move exclusively by either dimension. A semi-transparent preview of linearly transformed tracks is displayed during mouse drag

[17] A sound here specifically means a set of tracks that are often edited together.

events. The linear transform can be expressed for each selected spectral peak as simply:

$$\begin{aligned} p_f &+= d_f \\ p_t &+= d_t \end{aligned} \tag{2.2}$$

where p_f is the frequency argument of a spectral peak in Hz., p_t is the time argument of a spectral peak in seconds, and d_f and d_t are the frequency and time arguments of the mouse drag distance respectively.

2.4.4 Time Stretch

Selected tracks can be time stretched by clicking and dragging on the left or right edge of the bounding box of a track selection. The mouse cursor displays a left/right arrow when placed over the area of a bounding box that allows time stretching during mouse move events. During mouse drag events, a semi-transparent preview of the altered tracks is displayed. Time stretching linearly compresses or expands peaks in time relative to the first or last peak in time of all selected tracks. Time stretching for each selected peak is expressed as:

$$p_t += \frac{m_t - q_t}{d} \tag{2.3}$$

where m_t is the time argument of the mouse drag, q_t is the time argument of the anchor peak,[18] and d is the duration of the track selection. The time stretch feature may be used to reverse time for selected tracks too. The reverse effect is a useful artifact of the time stretch algorithm, and happens if the user drags the edge of the bounding box past the opposite edge.

[18]The anchor peak is the either the first or last peak of the track selection in time. When the user drags the right side of the bounding box, the first peak is the anchor, and inversely.

2.4.5 Pitch Shift

Selected tracks can be pitch shifted by clicking and dragging on the top or bottom edge of the bounding box of a track selection. The mouse cursor displays an up/down arrow when placed over the area of a bounding box that allows pitch shifting during mouse move events. During mouse drag events, a semi-transparent preview of the altered tracks is displayed. Pitch shifting proportionally translates peaks in frequency relative to the highest or lowest peak in frequency of all selected tracks. Pitch shifting for each selected peak is expressed as:

$$p_f *= \frac{q_f - m_f}{h} \tag{2.4}$$

where q_f is the frequency argument of the anchor peak,[19] m_f is the distance of the vertical mouse movement in Hz., and h is the height of the track selection in Hz. The harmonic equivalent of time stretch, *harmonic rotation*, is not an artifact of the pitch shift operation and is therefore not currently implemented in AudioChalk.

2.4.6 Duplicate

Any movement operation performed on selected tracks—linear transform, time stretch, or pitch shift—can also be used to create duplicates of those tracks. Track duplication is performed by holding **alt** during any of the three basic movement operations. The mouse cursor displays a + sign when holding **alt** while moving tracks to show that transformed tracks will create a duplicate. When a track is duplicated with the *alt* modifier, the *frequency* and *time* deltas are saved. Saved duplicate deltas can be repeated rapidly by pressing **cmd** + **d** while one or more tracks are selected. This means that, for example, regular harmonics or rhythms can be quickly generated from a few mouse clicks and a keyboard shortcut. An example of using the duplicate feature to quickly

[19] The anchor peak is the either the highest or lowest peak of the track selection in frequency. When the user drags the top side of the bounding box, the lowest peak is the anchor, and inversely.

make many tracks from one is demonstrated in Figures 2.6 through 2.8 below.

Figure 2.6: Duplicating tracks with **cmd** + **d** is an efficient way to generate a lot of musical material quickly.

Figure 2.7: For example, track duplication can quickly generate simple but dense harmonic structures.

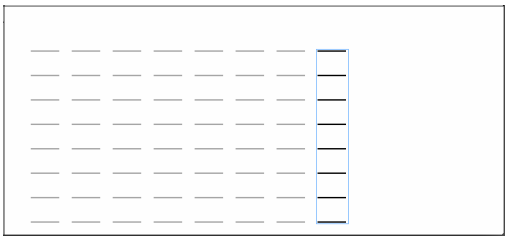

Figure 2.8: And it can quickly generate simple rhythmic patterns too.

Of course, using the duplicate function to build regular harmonic series and rhythms often sounds artificial. Most naturally occurring sounds and other musical sounds have irregular harmonic structures and rhythms. The swappable preferences window, *select editor*, has three sliders called *duplicate time*, *duplicate frequency*, and *duplicate gain*, which affect the *momentum* of the duplicate command. Momentum is a scaler that compounds the saved duplicate value on each iteration. For example, if

the frequency momentum is set to below 1, the frequency delta of repeated duplicate commands will decrease steadily, and inversely. Using the duplicate command with the momentum sliders is demonstrated in Figures 2.9 and 2.10 below.

Figure 2.9: Decreasing vertical momentum with the duplicate command.

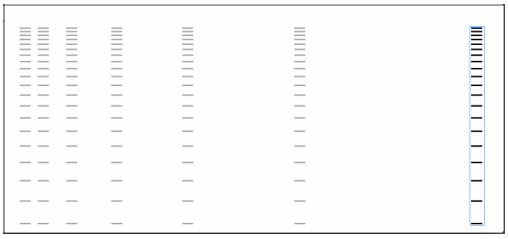

Figure 2.10: Increasing horizontal momentum with the duplicate command.

Naturally, even more complicated structures can be precisely and quickly generated by fiddling with the sliders while taking advantage of both the **cmd** + **d** shortcut. Another layer of complexity can be added to this technique by incorporating track groups and the track selection options described above.

2.4.7 Swappable Preferences Window

AudioChalk uses an integrated preferences window for users to quickly change the state of their session and of other important functions.

The swappable preferences window let's users control parameters for *track selection*, *audio importing*, and *session preferences*. The three individual preferences windows are kept separate for two reasons. First, it minimizes the number of parameters that users have to comprehend at once. The speed of finding the parameter that the

user wants to change is increased by having less parameters on screen simultaneously. Since preferences windows are separated by task, it's rare that users will have to change between preferences windows very frequently. However, to further increase the speed of navigating through all the available parameters, certain actions automatically open relevant preferences windows. For example, selecting the import audio command automatically opens the import audio preferences window. This way, users will rarely have to manually choose their intended window from the pulldown menu.

2.5 Rendering Audio

AudioChalk handles audio rendering for two purposes. First, audio can be rendered in-application and played through local hardware for rapid auditioning. Second, audio can be exported, or written to disk as one of a few standard audio file formats. In either case, the process of rendering audio from tracks is the same: audio is rendered with respect to the export settings, which are derived from the session preferences window described in the previous Section.

Recall from Section 2.2.1, *Anatomy Of A Track*, that tracks are made of 1 to n track segments. Also that track segments generate linear sinusoidal chirps given the parameters of their surrounding spectral peaks, which define a start and end point for time, frequency, and gain. Equation 2.5 expresses how track segments are used to generate linear chirps when $x(t)$ is the signal of the linear chirp over time t, $a(t)$ is the amplitude curve of the track segment, $\omega(t)$ is instantaneous radial frequency, and ϕ is the initial phase, which by default is 0.

$$x(t) = a(t) \cdot \sin(\phi + \omega(t)) \tag{2.5}$$

The instantaneous radial frequency of the oscillator is expressed as:

$$\omega(t) = f_0 + k \cdot t \tag{2.6}$$

where f_0 is the frequency in Hz. of the starting peak of the track segment and k is the rate at which frequency changes, which is expressed:

$$k = \frac{f_1 - f_0}{T} \tag{2.7}$$

where f_1 is the frequency of the ending peak, and T is the duration of the segment.

The gain curve of the track segment is defined by more parameters than the instantaneous frequency. The gain curve is derived from both *segment-scope* variables with duration t, and *track-scope* variables, with a duration of L. Similarly to how all samples of track segments from 0 to n are expressed as t, all samples of a track from 0 to n are expressed as l.

The gain of a track segment can be affected by the *constant gain c* of a track, the *fade-in* and *fade-out* times of a track $m(l)$, the track segment's peaks $p(t)$, and the *track envelope* $n(l)$. Therefore, $a(t)$, from equation 2.5, is expressed as:

$$a(t) = c \cdot n(l) \cdot m(l) \cdot p(t) \tag{2.8}$$

The amplitude envelope $n(l)$ is calculated from the loudness automation curve that tracks inherit when they are created. A track's loudness curve is an array of between 2 to n points with X-Y coordinates, sorted in ascending order by X values, and where X and Y values must both lie between 0 and 1. All new tracks, including tracks that are imported, have a loudness curve set to constant unity gain by default unless otherwise

specified. The contents of their unity gain arrays are two points $[0,1]$ and $[1,1]$. When a track is rendered to audio, the X value of each point is converted to time by multiplying its value by the duration of the track in samples, and rounding to the nearest sample, as expressed in equation 2.9.

$$n_x(l) = \lfloor L \cdot n_x(l) + 0.5 \rfloor \tag{2.9}$$

Y values of the loudness curve are already valid gain values, and are exponentially interpolated to create the loudness curve, n(l), as expressed in equation 2.10.

$$n_y(l) = \left((1 - \frac{l}{B}) * n_y(l) + \frac{l}{B} * n_y(l+1) \right)^2 \tag{2.10}$$

where B is the duration of the breakpoint segment.

Fade in and fade out amplitudes are expressed as:

$$m(l) = \begin{cases} \sqrt{\frac{l}{M}} & \text{fade-in if } l < M, \\ \sqrt{\frac{M-L-l}{M}} & \text{fade-out if } (L-l) < M, \\ 1 & \text{else.} \end{cases} \tag{2.11}$$

where M is the fade duration in samples.

Finally, the segment-scope gain scaler $p(t)$ is expressed:

$$p(t) = v_0 + j \cdot t \tag{2.12}$$

where v_0 is the gain scaler of the starting peak of the track segment, and j is the rate at which gain changes, which is expressed:

$$j = \frac{v_1 - v_0}{T} \tag{2.13}$$

When tracks have more than one track segment, phase and gain arguments are saved to prevent discontinuous jumps in a track's waveform.

2.6 AudioChalk: Conclusion

All of AudioChalk's features have been described, as well as hints about how AudioChalk is intended to be used. However, any techniques suggested are only just suggestions. The feature set described in this Chapter is intended most importantly to be open ended. Hopefully, future users will be able to develop their own styles of working with AudioChalk that best suit their compositional needs, regardless of how well I, the developer, was able to anticipate them. AudioChalk will be an ongoing project, and plans for possible features and changes will be discussed more in the final Chapter, *Future Work And Conclusion*.

The development rubric used to guide decisions about AudioChalk's design, *idiosyncrasy*, is described in detail below.

Chapter 3

Idiosyncrasy

Since software is deterministic, applications can only simulate implicit communication. Any rules about how an environment should handle implicit instructions must be created at compile-time by the developer.[1] This means that implicit-like communication between a composer and their software depends on assumptions made by the software developer. While developers have generated a lot of useful software by making assumptions about users intentions, this same practice limits the types of communication a user can have with their software. At least that is the general case. When developing software for a specific type of user, for example the experimental composer, *some* types of assumptions can be made with very little negative impact on the environment's scope. Though software is still more flexible when less assumptions are made about user's intentions, pre-compile-time assumptions that have minimal impact on scope increase the *idiosyncrasy* of that software. In other words, the scope of the software is limited, but not necessarily in a way limits the possible types of work for intended users. How can a developer gauge the impact on scope that assumptions they make will have? Defining idiosyncrasy as a rubric for developers intends to give insight to that question.

[1] Rules about implicit communication that users define are possibilities created by the developer at compile-time, and are not any different.

3.1 Defining Idiosyncrasy

Pragmatic solutions to user-centric software design can be arrived at by considering the following criteria: *usability*, *efficacy*, *intuitiveness*, and *flexibility*. These terms define the criterion idiosyncrasy, which itself describes the relationship between qualities of an environment's workflow from a user's perspective with respect to the environment's scope. To define idiosyncrasy as a rubric, first the components of experimental electronic composition environments must be defined, followed by the four basic criteria.

3.2 Components Of Environments

Developers of creative environments must consider the operation of environments as the product of three individual conceptual components: the environment's *interface*, *interpreter*, and *tools*.

The *interface* is an environment's canvas on which a user creates and manipulates a specialized, or idiosyncratic representation of musical metadata. For example, the interface of a DAW includes tracks on which time-domain representations of audio are manipulated. Other examples are the canvas of a patcher, on which users create, move, and connect objects, messages, and atoms; and the text editor of a specialized coding language with which a user makes and edits their code.

The *interpreter* is a set of rules that determine how data is represented on the interface, and how data from the interface affects an environment's tools. For example in a DAW, users can choose to increase vertical zoom in order to see a higher resolution representation of amplitude data in their tracks. Increasing the vertical zoom however, does not increase the amplitude of the tracks. Conversely, a user may increase amplitude of a track in a DAW by adjusting a volume slider for a track, which typically does not alter the representation of the audio in a track. These decisions are handled by the interpreter.

Finally, *tools* are a collection of functions that are directly responsible for the synthesis and manipulation of the environment's output and input. For example, the gain function that manipulates audio data in any environment is a tool. Tools are invoked by the interpreter, which the user controls with the interface. The data that is subject to tools, whether output or input, is generally called *media*. Media can be either musical data (e.g. audio files) or metadata (e.g. automation).

3.2.1 Usability

Usability describes the amount of effort it takes a user to achieve a task [BD13]. Consider the composer whose task is to filter an audio file and save the results to disk. An environment that takes a minimum of 15 clicks to accomplish the task is less usable than another environment that takes 7 clicks. In other words, usability compares the efficiency of the various input methods available to creative paradigms toward accomplishing a task. The criterion usability most directly describes the efficiency of an environment's interface.

3.2.2 Efficacy

Efficacy describes the impact of interpreter errors on an environment's output. To isolate efficacy as an idea, one should temporarily assume that the user and the environment's tools are infallible, and then measure the error of an environment's output for a given task based on expected results. The presence of errors in this scenario detracts from the efficacy of an environment. There are two types of operations common to any compositional environment that affect its efficacy: representational and interpretational errors [Cou04].

For example, errors in the representation of media on an environment's interface

will lead to the presence of artifacts in the output. Suppose that audio is presented to the user as editable in the time-frequency domain. Inaccuracies in the representation of the audio—say from interpolated representation of some parameter that is a higher resolution than the environment's tools can generate—will lead to unintended artifacts in the output that the user cannot repair. Conversely, an environment's inability to correctly interpret a user's input also leads to the presence of unintended artifacts in the output. Consider the case of audio synthesized by a user in the time-frequency domain. Quantization error—say from the width of frequency bins—may contribute to errors in the spectral content of the output regardless of the exactness of input from a user.

3.2.3 Intuitiveness

Efficacy by itself is not a function of usability. The relationship between usability and efficacy is described by another criterion called intuitiveness [Ham02]. Intuitiveness describes the difference between a user's expectation of a program's output given the user's input. Consider a user who wants to edit an interleaved stereo audio file. Most DAWs are optimized for editing interleaved stereo audio files. One could describe editing stereo audio in DAWs as rather intuitive. Contrarily, patchers and musical coding languages often require users to conjure magical syntax to import, edit, and export multichannel audio. Although DAWs are not necessarily either more efficacious nor more usable than patchers or languages for multichannel processing, they are certainly more intuitive.

3.2.4 Flexibility

Flexibility describes the potential scope of an environment's output, including in some cases output that the environment's original developer did not anticipate [HTS+07].

Simply put, an environment is more flexible than another if it can be used to accomplish more tasks. The criterion flexibility conditionally depends on the criterion intuitiveness. Intuitiveness affects flexibility if and only if the environment accommodates open-ended user workflows, which are called sandbox environments. The design of a sandbox environment emphasizes the transparency of its interface and interpreter. This way, the user is encouraged to build an interpretive relationship directly with the environment's tools [MF97]. A user may find that a close-ended environment more efficiently handles tasks for which the environment was intended. Yet in this case, the criterion flexibility is independent of the criterion intuitiveness because a close-ended environment's flexibility does not depend on the quality of the environment's interpretive relationship with the user; the flexibility of close-ended environments is always static. Contrarily, as the composer gains an understanding of the inner workings of a sandbox environment, they will begin to understand the environment as a collection of tools that support interpretive interaction. In other words, the composer may begin to envision novel use cases for an environment. This type of interaction is described by the criteria flexibility with respect to intuitiveness.

3.2.5 Idiosyncrasy

These criteria, *usability*, *efficacy*, *intuitiveness*, and *flexibility*, describe the degree to which an user's workflow in an environment is efficient for a task. These criteria also describe workflow tradeoffs that software developers must consider: the quality of an environment's supported workflows cannot be optimized for all tasks. The way in which criteria are skewed for various tasks is called an environment's idiosyncrasy. In general, the workflows of highly idiosyncratic environments are more intuitive and less flexible. Accordingly, less idiosyncratic environments are less intuitive and more flexible. Take for example the language SuperCollider, which has a considerably narrower scope

than, and is also more idiosyncratic than C code. A composer's workflow is improved by using SuperCollider if and only if their task is within SuperCollider's scope; that is, if SuperCollider is flexible enough to accomplish the user's task. Generally, it is more efficient to use the most idiosyncratic environment to accomplish a task that falls within the scope of multiple environments. Thoughtful software developers may empower composers by developing environments whose idiosyncrasies interfere minimally with composers' creative intentions. Of course, this is more easily said than done.

3.3 Idiosyncrasy: Conclusion

Part of the challenge of developing open-ended, robust software environments for composers—especially environments that are optimized for a variety of tasks—is the lack of a formalized vocabulary with which to discuss design tradeoffs. The complexities of software design can be simplified to some extent by considering the impact of an environment's components in terms of an optimization problem with respect to the criterion idiosyncrasy.

The primary benefit that idiosyncrasy as a rubric was intended to serve is the evaluation of and guidance in making or improving novel paradigms of experimental software applications.

Chapter 4

Conclusion

In this thesis, I have presented my research on the question of how software developers can optimize musical content creation workflows. This research was realized in two ways. First, a software design rubric called *idiosyncrasy*, which is a generalized account of my observations of how other software applications that pushed conventional workflow models either succeeded or failed. Second, a software application called *AudioChalk*, which was an attempt to realize a new workflow model by interpreting the rubric I developed, or in other words, to test the hypothesis that idiosyncrasy posited.

To reiterate, AudioChalk is *not*, in and of itself, intended to solve any problems mentioned either in this thesis or by the experimental electronic composition community at large. Rather, it is one of hopefully many continued attempts to empower experimental composers by offering new methods of expression.

Future Work

AudioChalk is a proof of concept musical drawing tool, and is was developed to explore new strategies for improving the workflow and scope of compositional software applications. This, the first version of AudioChalk, is transitioning out of the alpha

stage of development into beta at the time of writing. Though, because AudioChalk is a research project and not a commercial product, there are a number of features that will continue to be added over time.

Future versions of AudioChalk plan to add support for multichannel audio. This includes the ability to import, export, and manipulate multichannel audio without drastically complicating the current style of use that AudioChalk encourages.

The current version of AudioChalk only supports linearly scaled frequency on its canvas. This greatly sped up the development process, but a logarithmically scaled canvas would be much more useful to most composers. Also, the option to switch between a linear and logarithmically scaled canvas would benefit the widest variety of users.

Regarding authoring tracks, users should be able to generate both linear and logarithmic track segments. Other custom segment curves could include *n*th order bezier curves, circular segments, and bands of noise that control subtractive synthesizers could be useful as well.

Regarding the manipulation of tracks, there are a number of musical features that could be added. For example, giving users the option to rotate tracks or groups of tracks along any two musical dimensions: (*time-frequency*, *time-gain*, and *frequency-gain* rotation). For groups of selected tracks, harmonic rotation would be a good trick too, though it would require remapping tracks which may or may not be beneficial to users. Spectral convolution is another neat trick that could be incorporated.

Finally, simultaneous pitch shift and time stretch would be a useful feature. Other than the obvious control method of grabbing bounding box corners with the mouse, the line and pencil tools could be used to trace the fundamental pitches of sounds to achieve a combined time and pitch shift.

Special Acknowledgements

AudioChalk owes a lot to the hard work of other dedicated coders and musicians, whose work is not cited above, but strongly influenced its development.

JUCE is a massive open source C++ library for rapid software development, and is also a great teaching tool. Jule's effort over the last decade, single-handedly assembling this codebase, certainly deserves mention [Ltd14].

The fftw library, or the Fastest Fourier Transform in the West, assembled by Matteo Frigo and Steven G. Johnson of MIT was used for all of the spectral code in AudioChalk [FJ14].

Inspiration for the track tracing algorithm is from two notable works, Dan Ellis' ExtracTrax [Ell11] and Michael Klingbeil's SPEAR [Kli09].

Other notable musical drawing tools include Iannis Xenakis' UPIC [Nun14], Thomas Baudel's HighC [Bau07], IRCAM's AudioSculpt [IRC10], and UI Software LLC's MetaSynth [LLC14].

Bibliography

[Bau07] Thomas Baudel. HighC, 2007.

[BD13] Rodolphe Bourotte and Cyrille Delhaye. Learn to think for yourself: Impelled by upic to open new ways of composing. *Organised Sound*, 18(2):134–145, 2013.

[Cou04] Pierre Couprie. Graphical representation: an analytical and publication tool for electroacoustic music. *Organised Sound*, 9(1):109–113, 2004.

[Ell11] Dan Ellis. Sinewave and sinusoid+noise analysis/synthesis in matlab, 2011.

[FJ14] Matteo Frigo and Steven G. Johnson. Fftw: The fastest fourier transform in the west, 2014.

[Ham02] Michael Hamman. From technical to technological: The imperative of technology in experimental music composition. *Perspectives of New Music*, 40(1):92–120, 2002.

[Hic89] Leo Hickey. *The Pragmatics of style*. Routledge, 1989.

[HTS+07] Steve Harrison, Deborah Tatar, Phoebe Sengers, Steve Harrison, Courtesy Art, Deborah Tatar, and Phoebe Sengers. The three paradigms of HCI. In *In SIGCHI Conference on Human Factors in Computing Systems*, 2007.

[IRC10] IRCAM. AudioSculpt, 2010.

[Kli09] Michael Kateley Klingbeil. *Spectral Analysis, Editing, and Resynthesis: Methods and Applications*. PhD thesis, Columbia University, 2009.

[LLC14] UI Software LLC. Metasynth 5, 2014.

[Ltd14] ROLI Ltd. JUCE: Jule's utility class extensions, 2014.

[MF97] Wendy E. Mackay and Anne-Laure Fayard. Hci, natural science and design: A framework for triangulation across disciplines. In *Proceedings of the 2Nd Conference on Designing Interactive Systems: Processes, Practices, Methods, and Techniques*, DIS '97, pages 223–234, New York, NY, USA, 1997. ACM.

[Nun14] Alex Di Nunzio. UPIC, 2014.

[SJS90] Xavier Serra and III Julius Smith. Spectral modeling synthesis: A sound analysis/synthesis system based on a deterministic plus stochastic decomposition. *The MIT Press*, 14(4):12–24, 1990.

[ZÏ1] Udo Zölzer, editor. *DAFX: Digital Audio Effects*, volume 2. John Wiley and Sons, Ltd., 2011.