2014年度

博士論文

Affect-Based Aesthetic Evaluation and Development of Abstractions for Rhythm in Live Coding

多摩美術大学大学院美術研究科

ベル・ウッドサン・レニック

2014年度

博士論文

主査　久保田 晃弘 教授

副査　濱田 芳治 教授

Affect-Based Aesthetic Evaluation and Development of Abstractions for Rhythm in Live Coding

多摩美術大学大学院美術研究科

ベル・ウッドサン・レニック

Affect-Based Aesthetic Evaluation and Development of Abstractions for Rhythm in Live Coding

Renick Bell, Tama Art University

February 27, 2015

Contents

1 Preface

This document is based on previously published papers by this author, and it includes many passages largely unaltered from these papers which I have authored (or co-authored, in the case of the next-to-last paper in the list).

- An Interface for Realtime Music Using Interpreted Haskell, Linux Audio Conference 2011
- An Approach to Live Algorithmic Composition using Conductive, Linux Audio Conference 2013
- A Live Coding Improvisation, Association for Computing Machinery 9th Conference on Creativity and Cognition, 2013
- Towards Useful Aesthetic Evaluations of Live Coding, International Computer Music Conference 2013
- Pragmatic Aesthetic Evaluation of Abstractions for Live Coding, Japanese Society for Sonic Arts 17th Research Meeting, 2013
- Considering Interaction in Live Coding through a Pragmatic Aesthetic Theory, SI13: NTU/ADM Symposium on Sound and Interactivity, 2013
- Pragmatically Judging Generators, Generative Art Conference 2013
- Experimenting with a Generalized Rhythmic Density Function for Live Coding, Linux Audio Conference 2014
- (with Oliver Bown and Adam Parkinson) Examining the Perception of Liveness and Activity in Laptop Music: Listeners' inference about what the performer is doing from the audio alone, New Interfaces for Musical Expression conference, 2014

- Considering Interaction in Live Coding through a Pragmatic Aesthetic Theory, eContact!

2 Summary

Live coding, the interactive control of algorithmic processes through programming activity, is a relatively new method for the real-time production of media works. Such works are often musical or sound art performances. One principle tool in programming is abstraction, which refers to the process of aggregating or generalizing to achieve a representation of something in order to make programming easier, as well as particular representations achieved through such means. A key component of music is rhythm, which is perceived through the time which elapses between the beginnings of audible events in the music. In live coding, a performer uses abstractions to control the rhythm of the music.

This research asks if it is possible to make well-reasoned evaluations of the abstractions used in live coding with the purpose of improving them. It further asked if such is possible, how those evaluations can be carried out. This paper hypothesizes that such evaluations are possible through a pragmatic theory of aesthetics drawn from the writing of John Dewey. This 20th century American philosopher has exerted influence on the development of aesthetics by pursuing an intuitive theory of art as experience. In addition, his writings on valuation can be applied to form a practical method for evaluation of such experiences.

A revised version of Dewey's theory can be stated as follows. An affect is an emotional state. An affectee is a person experiencing affects in an interaction with affectors. An affector is an object of perception that stimulates affects in an affectee. A work of art is an affector which in some way was created, organized, or manipulated with the intention of it being an affector. A person involved with the creation or arrangement of an affector is an artist. An art experience is the experience of affects in an affectee as the result of the affectee's

interaction with a set of affectors, with at least one of those affectors being a work of art. The art experience is the experience of those affectors either simultaneously or in sequence. Experience involves a possibly infinite number of affectors arrayed in a network in which they influence each other and affectees either directly or indirectly. Changing the network of affectors changes the experience. The value of an affector is connected to the value of the art experience in which it is involved. The value of an art experience is determined by the affects experienced and how well those affects and the other consequences of the experience suit the intentions of the affectee.

This aesthetic theory can be applied to an art experience in an analytical way using the following heuristic:

1. analyzing the intentions held by an affectee
2. determining the network of affectors that are present in the experience
3. examining the consequences of the interaction with those affectors, including resulting affects
4. determining the relationship between those consequences, affects, and intentions
5. either (a) assigning value to the experience and its affectors according to those intentions or (b) changing intentions and then repeating this process with the new intentions

The experience of live coding can be evaluated within such a framework. Abstractions, including those used for rhythm in live coding, function as affectors. It is also necessary to consider the other affectors involved, such as projection of the coding activity, the sound system, the musical features of the audio, the programming tools through which the perfor-

mance is realized, and the various styles of interaction with the live coding system. After examining the influence of the abstractions on the experience, a performer can make reasonable evaluations and apply the results towards improvements of those abstractions.

The intentions of the performer form the initial standard of evaluation. Live coders express a wide variety of intentions, such as flexibility of expression, discovery of new musical structures, or deeper interaction with computers when creating media.

Using the above method, it is possible to examine the live coding activity of this author, showing evaluations done over the period of this doctoral study and their results.

This author performs by generating rhythm patterns and their variations using a custom collection of abstractions for live coding called Conductive and the Haskell programming language. Several concurrent processes trigger a custom sampler according to those generated rhythms. Through interactive programming, the performer manages those concurrent processes and generates and selects data to be read by those processes, resulting in musical output. Performances are projected.

This author was seeking a freedom of expression through rapid change of rhythm patterns. Performances done with an early version of Conductive lacked the facility for carrying out coherent rhythm pattern changes quickly. These negative affects – feelings of being stationary or confusion from excessive randomness – impeded positive evaluation, thus requiring changes to Conductive.

The following version of Conductive contained new abstractions allowing the patterns to be changed more rapidly and producing variations bearing more resemblance to one another. The affects of stasis and chaos were diminished and a more propulsive groove began to

emerge in performances. However, an additional intention was to present a variety of rhythmic feels. At that time, Conductive did not allow the rhythmic feel to be changed easily, causing a different kind of rigidity to be perceived.

To remedy this, the structure of the existing abstractions was revised to include a new class of abstractions. It became possible to create abstractions for different types of rhythm patterns as well as methods for determining how their variations at different rhythmic densities would be produced. Some of these new type of abstractions were implemented, including a pattern generator based on the growth of plants. Live coding performances after those changes stimulated sensations of energy and had a more vigorous and sinuous quality.

Using the pragmatic aesthetic theory to analyze experience of live coding through the affects that result leads to incremental improvement in the underlying affectors – abstractions for rhythm – and consequently experiences of live coding performances involving those abstractions.

In conclusion, this research contributes the following:

1. a definition of abstraction and an argument for its centrality to live coding
2. some music theory and techniques for the generation of rhythm patterns
3. some tools for the generation of rhythm patterns in the Haskell programming language based on the music theory and techniques mentioned above
4. a series of performances, both domestic and abroad
5. a restatement of Dewey's aesthetic theory, leading to an affect-based pragmatic heuristic for the evaluation of art experiences
6. an analysis of the many facets of live coding

7. an argument that abstractions can be evaluated aesthetically
8. applications of the pragmatic heuristic to live coding, in particular abstractions for rhythm in live coding

3 Introduction

In a performance space, a performer is standing in front of a table with a laptop computer on it. On the wall behind the performer, the code for a computer program can be seen through a projection from a projector attached to the performer's laptop. Large speakers in the corners of the room emit loud electronic sounds including drum sounds that are a result of the displayed code. The performer makes changes to the code, which the audience can see, and then with a press of a button the code runs and the sound changes in complex ways. The audience watches the changing code, sometimes still, sometimes nodding their heads, and sometimes dancing to the rhythms coming out of the speakers. This activity, called live coding, is a relatively new media art practice.

Live coding is not a genre of music. Rather, it is a technique which is used for audiovisual performances in a variety of genres, including sound art that some people might not consider music. It traditionally involves the projection described above, and it is also usually improvised. Performances can involve multiple performers or be carried out solo.

I have been involved in this practice since 2007, with a first public presentation of my live-coding work in 2011. My practice involves performances and software in the form of a library called Conductive. In that practice, there have been numerous occasions on which I have encountered the question of how to evaluate live coding.

For example, an engineer asked me how to judge the merit of my software. He wanted to see numeric data which demonstrated the value of the library. Psychological testing, which can be quantitative, would be acceptable. The problem would then be the content of such tests.

When presenting work at the Linux Audio Conference in 2011, I asked the audience for advice on obtaining quantitative judgments of such software. The feedback they gave showed they were skeptical of the potential for quantitative measurement, with audience members identifying the tool as a musical instrument. The feeling that instruments such as basses and bassoons cannot be compared in a way that makes it possible to say that the bass is a better instrument than the bassoon or vice versa.

Comparisons can be made of the type that say the bassoon is a better instrument for a particular piece because the composer desires its unique timbre, or that the bass is a better instrument for a player because it fits the player's physical attributes more closely. One judgment is based on compositional factors, and another based on an interface issue.

While considering these issues, it became interesting to me to search for more fundamental principles that might be useful in both of the cases above. With such principles, it might become easier to evaluate live coding and the tools used in such performances. The field of aesthetics seemed to hold answers, so this research moved in that direction.

3.1 An Outline of this Document

This document first introduces readers to the practice of live coding. After defining live coding, it explains how live coding is carried out and the justifications for doing so. It then presents the problem statement for this research. Before describing how the questions of the problem statement are answered, the background to this statement is explored, including a history of live coding, an examination of abstraction, a brief look at musical rhythm, and how these two issues are related in live coding. The document then explains the methods

for pursuing the problem statement, both theoretical research and practice-based research. That begins with music theory research on rhythm patterns and density, proceeds to the implementation of those theories in software, and sees practical application in live coding performances. Following the discussion of this creative work, a discussion of that work based on a pragmatic aesthetic theory takes place. This section briefly explains pragmatism and philosopher John Dewey's aesthetic theories. It then develops those ideas into a method for analyzing experiences aesthetically and a heuristic for evaluating such experiences. That method is applied to live coding tools and performances generally and then to my own live coding tools and performances. Based on that discussion, the conclusions from this research are made and answers to the questions in the problem statement are given. Finally, directions for related future research are described.

3.2 Definition of Live Coding

According to McLean, "the nature… of live coding is still not well established."[1] With that in mind, this paper proceeds to clarify what live coding is and consists of.

Collins presents two definitions of live coding:[2]

1. "Live computer music and visual performance can now involve interactive control of algorithmic processes. In normal practice, the interface for such activity is determined before the concert. In a new discipline of live coding or on-the-fly programming the

[1] McLean and Reeve, "Live Notation," p.70.

[2] Collins, "Live Coding of Consequence," p.207.

control structures of the algorithms themselves are malleable at run-time"[3]

2. "Digital content (music and/ or visuals predominantly) is created through computer programming as a performance"[4]

This paper defines live coding as the interactive control of algorithmic processes through programming activity as a performance, a definition derived from Brown, Collins, and Ward.[5] This paper will not consider programming in front of others as a tutorial or make a distinction between public performances and solitary private coding in a studio performance.

An algorithmic process is a process driven or controlled by an algorithm. An algorithm is, according to Wordnet 3.0, "a precise rule (or set of rules) specifying how to solve some problem".

3.3 How Live Coding Is Done

Live coding involves interaction with the following tools:

- abstractions
- libraries
- an editor
- an interpreter
- a synthesizer

[3] Ward et al., "Live Algorithm Programming and a Temporary Organisation for Its Promotion," p.243.

[4] Brown, "Code Jamming," p.1.

[5] Collins, "Live Coding of Consequence"; Ward et al., "Live Algorithm Programming and a Temporary Organisation for Its Promotion"; Brown, "Code Jamming."

- (traditionally) a projector

3.3.1 Abstractions

This document develops the following definition of abstraction:

1. (uncountable) Abstraction is the process of aggregating or generalizing to achieve a representation of something in order to make programming easier
2. (countable) an abstraction is a representation of something achieved through aggregation or generalization that allows instantiation of itself in order to make programming easier

For a full explanation of this definition and discussion of abstractions, see section 4.2.

3.3.2 Other Tools

A software library is a collection of abstractions which are designed to be reused to save the time and effort of the programmer. An editor is a computer program used for the writing and changing of source code (text). An interpreter is a computer program that reads source code passed to it by a user and executes that code immediately. A synthesizer is a sound generator that responds to triggers received from the interpreter. A projector connected to the programmer's computer is used traditionally to show the audience the programmer's interaction with the system.

3.3.3 Analogies to Traditional Music

As long as the analogies are not pushed too hard, the following parallels can be observed. Abstractions in live coding are like the notes, chords, symbols, riffs, themes and other song parts, and instruments which are used in traditional musical compositions and performances. These are close to the smallest units employed by live coders to perform. Software libraries are roughly equivalent to collections of riffs, sheet music, drum sets, ensembles, sheet music stores, and instrument stores. An editor can be thought of like blank staff paper and a pen, a conductor's stand and baton, or the interface of a physical instrument (keys, heads, reed, or other interface component). An interpreter functions like the internal mechanisms of an instrument, such as hammers and strings in a piano, or the orchestra from the conductor's viewpoint. Using a projector is like a traditional instrumental performer facing an audience on a stage so that they can see the playing.

The interaction in live coding can be compared to the activities of a traditional musician or composer. Selecting libraries is similar to choosing instruments and sheet music or arrangements. Preparing abstractions can be like preparing riffs or modifying instruments. The preparation of the performance tools and environment listed above is parallel to loading in the instruments and equipment for the concert. When a live coder loads libraries, it can be thought of as setting up the instruments and putting the sheet music on the stands. Live coders write code using abstractions, which corresponds to musicians picking up the instruments, warming up a horn with air, or thinking about riffs. Executing the code in the interpreter eventually causes the synthesizer to be triggered. That resembles the action of a traditional musician starting to play an instrument. Editing code to change the nature

of the events in the performance is crucial, and this can be compared to the improvisation, soloing, and adjustments to phrasing and dynamics that a traditional musician does in a performance. Just as a traditional musician stops playing, a live coder stops the execution of code to end a performance.

3.4 Justification for Live Coding

One of the drawbacks of live coding is the hard mental operations that it requires. For a more complete discussion of the usability issues involved in live coding, see Blackwell and Collins.[6] Another factor is the potentially slow text manipulation that live coding requires.[7] Brown explains that time constraints in performances require concisely-notated and appropriate abstractions to produce coherent non-static musical content.[8]

Despite these challenges, live coding has its benefits. Live coding enables a more abstract manipulation of a representation of music than physical gestures used for playing instruments. It is also thought to be more convenient in many regards than windows-icon-mouse-pointer software.[9]

From another perspective, it takes the potential of algorithmic composition and turns it into a live performance rather than a write/compile/run loop from traditional software development or electronic music composition. Seen this way, it can be thought of as an extension of

[6] Blackwell and Collins, “The Programming Language as a Musical Instrument.”

[7] Sorensen and Brown, “Aa-Cell in Practice,” p.3.

[8] Brown, “Code Jamming,” p.1.

[9] Bell, “An Interface for Realtime Music Using Interpreted Haskell,” p.1.

algorithmic composition practices.

Graphical sequencers are poor tools for live musical performance in the judgement of this author. Users interact with them primarily through a mouse, a limited number of keyboard shortcuts, or external hardware that requires extensive setup and lacks portability. These graphical applications allow limited customizations to the manner in which they are controlled. Previous experiences with GUI tools in performance showed them to be inflexible and awkward when trying to execute complex sets of parameter changes simultaneously.

This problem is exacerbated when considering the wide variety of synths which exist. Musicians would like to use them together freely, but coordinating them is difficult. For use with graphical sequencers, synths employ separate GUIs which are almost always point-and-click or require troublesome set-up to use with external hardware and thus cannot be easily manipulated simultaneously with other parameters.

One possible solution to this problem may be the use of a programming language as a tool for live coding of music.[10] A programming language with abstractions for music can serve as a control center for a heterogeneous set of outputs. The user can specify complex parameter changes to be executed simultaneously. A wide variety of existing programming tools and text-manipulation utilities can be used to make this process more efficient.

By producing media output through the manipulation of symbols which represent abstractions rather than physical, gestural activities, it seems possible that great potentials in expressivity are brought into being. Gesture may be more limited than manipulation of symbols representing abstractions. While humans have learned to use a complex vocabulary of ges-

[10] Collins et al., “Live Coding in Laptop Performance,” p.322.

tures to produce dance, art, and operate musical instruments, going past the limitations of gestures by using in realtime a more complex system of symbols may increase the range of what is expressible. The full potential of live coding has yet to be uncovered.

These justifications are personal ones held by the author. Other live coders may have other reasons for choosing live coding, such as not being able to play musical instruments or other reasons. There are many intentions that live coders pursue when live coding, and some of these are presented in the discussion section of this document (section 6.4.1).

Rather than theorize further about individual coder's justifications for using live coding techniques or the potentials of live coding (a potential which the practice related to this research hopefully demonstrates), this research pursues a different problem described in the problem statement below.

3.5 Problem Statement

A key component of music is rhythm, which is perceived through the time which elapses between the beginnings of audible events in the music. Some element of the program that a live coder is developing must control the timing, and consequently the rhythm, of the events. Abstractions are principle tools for doing so.

This research asks if it is possible to make well-reasoned evaluations of the abstractions for rhythm used in live coding with the purpose of improving them. It further asks if such is possible, how those evaluations can be carried out.

This paper hypothesizes that such evaluations are possible through a pragmatic theory of

aesthetics drawn from the writing of John Dewey on aesthetics and valuation.

This research attempts to test that theory through the development of abstractions for rhythm in live coding and then corresponding attempts at improving those abstractions based on aesthetic evaluation using that theory.

4 Background

4.1 A Brief History of Live Coding

Collins attributes ancient origins to live coding in the work by Guido d'Arezzo, who in the 11th century provided the foundations of modern musical notation. He finds a literary precedent for live coding in the "The Glass Bead Game" of 1943 by Herman Hesse. He notes mid-20th century avant-garde text scores like those of George Brecht along with the development of interactive computing as other precedents.[11]

The first documented live coding performance was carried out by Ron Kuivila at STEIM in 1985.[12] The piece, "Watersurface", was done in a "precursor of Formula", a programming language called the FOrth MUsic LAnguage developed by David Anderson and Kuivila,[13] and involved Kuivila "writing Forth code during the performance to start and stop processes triggering sounds through [a Mountain Hardware 16 channel oscillator board]."[14] "[The] original performance apparently closed with a system crash."[15]

From 1985 to the early 90s, The Hub used a standarized data structure which was shared via network and interpreted by the programs of each performer to carry out performances.[16]

[11] Collins, "Origins of Live Coding."

[12] Sorensen, "Impromptu," p.2.

[13] Anderson and Kuivila, "Formula."

[14] Kuivila, "A Performance at STEIM in 1985 in Which You Used the Formula Programming Language."

[15] Kuivila et al., "A Prehistory of Live Coding."

[16] Gresham-Lancaster, "The Aesthetics and History of the Hub," p.41.

Earlier (1980-82), the League of Automatic Composers did extended performances with programs which were tuned as they ran.[17] The degree to which such tuning required coding is unspecified in existing literature.

Scot Gresham-Lancaster cites John Cage and David Tudor as influences.[18] This impact is supported by Michael Nyman,[19] who contrasts experimental music like that of Cage, Steve Reich, and various performers belonging to the Fluxus group with what he calls the avant-garde. Nyman writes that experimental music focuses on situations in which processes work across a field of possibilities to bring about unknown outcomes[20] and show the uniqueness of particular moments.[21] Such music may have an identity located outside of the final audience-perceived auditory material,[22] have non-traditional methods for dealing with time,[23] and force performers to use skills not typically associated with musicianship.[24] Experimental music frequently presents performers with surprise difficulties in performance,[25] resemble games,[26]

[17] Brown and Bischoff, “Indigenous to the Net.”

[18] Gresham-Lancaster, “The Aesthetics and History of the Hub,” p.39.

[19] Nyman, **Experimental Music**, p.2.

[20] Ibid., p.3.

[21] Ibid., p.8.

[22] Ibid., p.9.

[23] Ibid., p.10.

[24] Ibid., p.13.

[25] Ibid., p.14.

[26] Ibid., p.16.

and give performers rules to interpret.[27] Who the performers are may be ambiguous.[28] It may require new ways of listening.[29] These issues appear in live coding frequently, such as the need for new performance skills[30] and the similarity to games.[31] One thing that appears to separate live coding from this experimental music tradition is that performers before live coding seem not to have changed the rules during performances, while this is a central concern for live coding. Further examination of the 20th century experimental music tradition may reveal additional interesting connections with live coding.

As mentioned in previously, live coding can also be considered a type of algorithmic composition, which has a history that could extend back as far as Ptolemy's music theory,[32] and certainly as far back as music dice games such as Mozart's Dice Music.[33] More modern examples of algorithmic composition practice include the twelve-tone music of Schoenberg,[34] computer music work by Caplin and Prinz followed by Hiller and Issacson,[35] the aleatoric

[27] Ibid., p.16.

[28] Ibid., p.19.

[29] Ibid., p.20.

[30] Brown, "Code Jamming," p.1.

[31] Magnusson, "Confessions of a Live Coder," p.6–7.

[32] Maurer, **A Brief History of Algorithmic Composition**.

[33] Hedges, "Dice Music in the Eighteenth Century."

[34] Schoenberg, **Fundamentals of Musical Composition**.

[35] Ariza, "Two Pioneering Projects from the Early History of Computer-Aided Algorithmic Composition," p.40.

music of Cage and Stockhausen,[36][37], Xenakis's stochastic music,[38] and the generative sequences made in Max on Autechre's Confield.[39] A full exploration of the history of algorithmic composition is beyond the scope of this document.

After an apparent gap in the 90s, live coding activity increased in the first decade of the 21st century.[40] Slub began performing laptop music with the content of their screens projected in 2000.[41] Julian Rohrhuber also managed to change running code in SuperCollider 2 at this time. In 2000, Rohrhuber and SuperCollider author James McCartney peformed at the International Computer Music Conference. In that performance, they traded code over a computer network. The sound programming language Chuck, with facilities for live coding, dates to 2003.[42] Alex McLean's article on "Hacking Perl in Nightclubs" appeared on Slashdot in 2004, in which he describes interacting with running Perl scripts to produce music, including a script that can edit itself[43] Other early examples include Julian Rohrhuber's experiments in live coding with SuperCollider and the duo of McLean and Adrian Ward (later joined by David Griffiths) called Slub.[44] Collins lists other early live coding milestones, such as the 2005 TOPLAP performance at the Berlin Transmediale and a first "live code battle" in

[36] Kostelanetz, **Conversing with Cage**.

[37] Paul, "Karlheinz Stockhausen."

[38] Xenakis, **Formalized Music**.

[39] Tingen, "Autechre, Recording Electronica."

[40] McLean, "TOPLAP Website."

[41] Collins, "Origins of Live Coding."

[42] Ibid.

[43] McLean, "Hacking Perl in Nightclubs."

[44] Collins et al., "Live Coding in Laptop Performance," p.323–26.

the same year. The duo aa-cell (Andrew Sorensen and Andrew Brown) began to practice in 2006, and the LOSS Live Code Festival was held in 2007. The BBC aired a documentary on pub code in 2009, and in 2013 a live coding festival was held in Karlsruhe, Germany.[45]

Since 2012, there have been an increasing number of algoraves, which are music events "where algorithms are explored in alliance with live electronic dance music" and in which live coding is frequently the method by which the music is performed.[46]

Systems used in the previous decade for live coding include McLean's feedback.pl, Griffith's Fluxus, and computer music languages SuperCollider[47] and Chuck.[48] It became common practice to project live coding activity for the audience to see during the performance.[49] McCartney, the creator of SuperCollider, said a specialized language for computer music wasn't necessary, but general purpose programming languages weren't ready yet practically.[50] Musicians managed to make music with domain-specific languages, but in the view of this author those are unsatisfactory in many ways, such as lack of libraries and development tools and slow performance. While those domain-specific languages continue to be used, additional systems like Extempore, Tidal, Overtone, and this author's Conductive have appeared, many of which now employ general purpose programming languages, and a growing culture and body of work has developed around live coding.

[45] Collins, "Origins of Live Coding."

[46] Collins and McLean, "Algorave."

[47] McCartney, "SuperCollider."

[48] Wang and Cook, "ChucK."

[49] McLean et al., "Visualisation of Live Code," p.1.

[50] Mccartney, "Rethinking the Computer Music Language."

4.2 Abstractions

WordNet 3.0 gives us the following definitions of abstraction.

1. a concept or idea not associated with any specific instance;
2. the act of withdrawing or removing something;
3. the process of formulating general concepts by abstracting common properties of instances;
4. an abstract painting;
5. preoccupation with something to the exclusion of all else;[51]

Definition 5 is irrelevant for us. Definition 4 differs considerably, with a complete comparison between live coding and work in other media left for a future research project. 1 and 2 are closer. 3 is the closest to abstraction in computer science, yet it does not completely explain the matter.

It may be helpful to consider the verb root, “abstract”:

3. consider apart from a particular case or instance (Wordnet 3.0)

O’Sullivan and Pecorino give this example:[52]

> “I am feeling sad right now.” “Sad feelings often follow tragedies.” … the first statement… refers to a particular person, a particular feeling, and a particular time. In the second statement, the speaker is abstracting from experience.[53]

[51] Princeton, “About WordNet.”

[52] in Chapter 1 section 3

[53] O’Sullivan, Stephen and Pecorino, Philip, “Ethics - an Online Textbook.”

This is the root of abstraction in computer science. However, abstraction in computer science doesn't have to represent knowledge about the world, and must be much more specific, among other things.

Arnheim, in describing abstraction in art, describes it as the construction of a concept that contains the essential characteristics of a given kind.[54] Abstraction in computer science follows this direction. However, it does not use a visual language but a highly structured (usually) textual language and allow instantiation of themselves.

The Collins English Dictionary definition of instantiation is as "(Philosophy / Logic) the representation of (an abstraction) by a concrete example."[55]

Abstraction is one of the fundamental concepts of computer science,[56] which uses them to simplify the design and development process.[57] The Free Online Dictionary of Computing defines abstraction as "[producing] a more defined version of some object by replacing variables with values (or other variables)."[58] Dijkstra says abstraction in computer science is not as much about ignoring details as capturing essential details. Abstraction is the tool that gives computer science algorithms and variables, and it "permeates the whole subject [of computer science]."[59]

[54] Arnheim, **Visual Thinking**, p.173.

[55] HarperCollins, "Instantiation."

[56] Turner and Eden, "The Philosophy of Computer Science."

[57] Keller, **Computer Science**, p.1.

[58] Howe, "Abstraction."

[59] Dijkstra, Dijkstra, and Dijkstra, **Notes on Structured Programming**, p.13–14.

Shaw reiterates:

> The essence of abstraction is recognizing a pattern, naming and defining it, and providing some way to invoke the pattern by its name without error-prone manual intervention… In other words, good abstraction is ignoring the right detail at the right times.[60]

Focusing on the domain of database design, Smith and Smith define two types of abstraction: aggregation and generalization. The former "turns a relationship between objects into an aggregate object." Generalization yields a generic object from a class of objects.[61] However, there would seem to be many abstractions that both aggregate and generalize. As an example, consider a multi-step procedure that is parameterized in order to cover a range of cases.

Abstraction in computer science can be divided into two categories according to what is abstracted: data abstraction and control abstraction, the latter of which includes procedural abstraction.

An analysis of abstraction in computer science leading to an interesting definition is given by Shutt.[62] He lists four primary categories of things researchers call "abstraction". Quoting Shutt, they are:

- Use of symbolic names for constants/variables.

[60] Shaw, "Larger Scale Systems Require Higher-Level Abstractions," p.143.

[61] Smith and Smith, "Database Abstractions," p.105–7.

[62] Shutt and Shutt, "Abstraction in Programming - Working Definition," p.7.

- Definition of procedures and functions (procedural or functional abstraction).

- Definition of new data types, especially when packaged with associated operations (data abstraction).

- Definition of new control structures (control abstraction)[63]

Shutt discusses other definitions, such as those based on generalization, the programming language as notational system, and parametric definitions. He finds them unable to account well for the four cases described above. He settles on this working definition:

> Definition 3.1 (in programming) Abstraction: The act or process of transforming one programming language into another by means of facilities available in the former language…[64]

He then describes two types based on this definition:

> Definition 3.2 Incremental abstraction: An abstraction that modifies a programming language gradually, by means of selective changes.
>
> Definition 3.3 Radical abstraction: An abstraction that replaces one programming language with another wholesale, by means of explicit computation.[65]

[63] Ibid., p.4–5.

[64] Ibid., p.7.

[65] Ibid., p.8–9.

A process called abstraction is described above. However, in definitions 3.2 and 3.3 above, Shutt begins to mention “an abstraction”. This shows how programmers also talk about “abstractions” as things that can be counted, manipulated, and used. In this sense, abstractions in computer science are components, which are like parts in machine, with which programs are constructed that hide the complexity of the foundations of the program. Abstractions consist of ways to organize data and processes to carry out using that data. They can be instantiated. They typically have parameters.

The definition found in Green and Blackwell is:

> An abstraction is a class of entities, or a grouping of elements to be treated as one entity, either to lower the viscosity or to make the notation more like the user’s conceptual structure[66]

They anticipate Shutt, writing “The characteristic of an abstraction… is that it changes the notation.[67]”

Blackwell, Green, and Nunn concisely explain the intent of abstractions:

> Diffuseness is a measure of how much or little can be said in a few word or symbols. If a notation is too terse, it encourages slips of the pen; if too diffuse, it takes too long to get anything done. Diffuseness can be reduced by abstractions… Abstractions wrap up several symbols into one, or do other similar jobs. They are ways to change the fundamental notation. They can

[66] Green and Blackwell, **Cognitive Dimensions of Information Artefacts**, p.24.

[67] Ibid., p.24.

> reduce diffuseness and increase clarity…[68]

The instantiation of abstractions is often phrased as “software reuse”, described by Krueger:

> Abstraction plays a central role in software reuse. Concise and expressive abstractions are essential if software artifacts are to be effectively reused. The effectiveness of a reuse technique can be evaluated in terms of cognitive distance – an intuitive gauge of the intellectual effort required to use the technique. Cognitive distance is reduced in two ways: (l) Higher level abstractions in a reuse technique reduce the effort required to go from the initial concept of a software system to representations in the reuse technique, and (2) automation reduces the effort required to go from abstractions in a reuse technique to an executable implementation.[69]

Abstraction in computer science is something that you probably touch daily. Think of the Google homepage. HTML is a set of abstractions for displaying web pages in browsers. An anchor tag, like this one, creates a link.

```
<a class=gbgt id=gbg6 href=" https://plus.google.com/u/0/me"  ...
```

The title tag places a title in the title part of the window.

```
<title>Google</title>
```

[68] Blackwell, Green, and Nunn, “Cognitive Dimensions and Musical Notation Systems,” p.3.

[69] Krueger, “Software Reuse,” p.1.

4.2.1 Definition of Abstraction

The following working definitions of abstraction are synthesized from various sources.[70] One aim is to capture the two uses of abstraction described above: both a process and an entity.

1. (uncountable) Abstraction is the process of aggregating or generalizing to achieve a representation of something in order to make programming easier
2. (countable) An abstraction is a representation of something achieved through aggregation or generalization that allows instantiation of itself in order to make programming easier

4.2.2 Layers of Abstraction

Abstractions can be layered. First, examine a daily-life example. Letters make up words. Words make up sentences. Sentences make up a written document. The layers of abstraction in a computer are similar to that. According to Keller:

> Thus we have the idea of implementing components on one level using components on the level below. The level below forms a set of abstractions used by the level being implemented. In turn, the components at this level may form a set of abstractions for the next level.[71]

[70] Smith and Smith, “Database Abstractions”; Shutt and Shutt, “Abstraction in Programming - Working Definition”; Green and Blackwell, **Cognitive Dimensions of Information Artefacts**; Keller, **Computer Science**; Dijkstra, “On the Role of Scientific Thought”; Shaw, “Larger Scale Systems Require Higher-Level Abstractions”; Krueger, “Software Reuse.”

[71] Keller, **Computer Science**, p.1.

It be said that there are low-level abstractions and high-level abstractions. Dijsktra recommends progressively refining the entities of a program in layers to ensure the correctness of the resulting program and describes program structure as “a layered hierarchy of machines.”[72] Building software in layers is recommended by Brooks.[73]

Simonyi defines higher level abstractions as those covering the “essential qualities of the program” that directly work toward the targeted solution. These are a separate category from accidental detail (a phrasing he borrows from Brooks), which includes things changeable for reasons such as optimization and compatibility. Even higher-level abstractions can be developed which can handle two different but similar problem statements, leading to increased code reuse.[74]

Consider an example from live coder Andrew Sorensen:[75]

```
(play-note (*metro* time) piano p
           (+ 50 (* 20 (cos (* pi time))))
           (*metro* ' dur dur1))
```

The abstraction “play-note” is a function with parameters. Other abstractions are used in parameters, such as the “cos” function. This function with parameters could be an abstraction if it were named, and in some usage cases it might be more useful if it could be called with one or more parameters.

[72] Dijkstra, Dijkstra, and Dijkstra, **Notes on Structured Programming**, p.64.

[73] Jr, **The Mythical Man-Month**, p.212.

[74] Simonyi, “Intentional Programming,” p.2.

[75] Sorensen, “A Study In Keith.”

4.3 Generators

Live coding is a type of generative art. In programming generative art, one class of abstractions describes generators, which are also called generative processes.

The definition of generative art from Galanter is:

> Generative art refers to any art practice where the artist uses a system, such as a set of natural language rules, a computer program, a machine, or other procedural invention, which is set into motion with some degree of autonomy contributing to or resulting in a completed work of art.[76]

Though McLean and Wiggins write that live coding differs from generative art in that "generative art is output by programs unmodified during execution„"[77] Galanter does not expect complete autonomy from generative art processes, just that the artist relinquishes at least partial control to the generative system.

Brown describes live coding as a method for interacting with generative processes, plainly classifying it as a member of the set of generative art practices:

> … generative music in an expansive sense [is substantial musical outputs which] are produced by an algorithm… It is the ability to harness generative material that allows live coding performers to participate in a new kind of performance where they exercise indirect, or meta, control over the creation of

[76] Galanter, "What Is Generative Art?" p.4.

[77] McLean and Wiggins, "Live Coding Towards Computational Creativity," p.1.

> their music.[78]

Brown describes the performer's position as:

> The performer is directly embedded within the algorithmic process and is free to guide and directly manipulate the unfolding of processes over time. The generative process exists on two levels, the playing out of the algorithmic potential of the code and the unfolding of the algorithmic opportunities and structural pathways held in the mind of the performer.[79]

Another paper predating the Brown reference above "[advocates] the humanisation of generative music."[80] That humanisation does not preclude the inclusion of live coding as a type of generative art.

Soddu's definition gets at the practical consequences of generative art: "construction of dynamic complex systems able to generate endless variations."[81] Live coding seems to fit this description well. This leads towards consideration of the aesthetics of generative processes in live coding, about which Collins writes that "… generative music is best appreciated when studied closely, when run many times…" and further asks "At a live concert, is generative music a music that says this time is special, now is privileged?"[82] These aesthetic questions are considered below.

[78] Brown and Sorensen, "Interacting with Generative Music Through Live Coding," p.3.

[79] Ibid., p.7.

[80] Ward et al., "Live Algorithm Programming and a Temporary Organisation for Its Promotion," p.246.

[81] Soddu, "Generative Art International Conference."

[82] Collins, "Generative Music and Laptop Performance," p.71.

4.4 Interaction

The practice of live coding can emphasize the interaction of the programmer with the software running on the programmer's computer. Interaction consists of various aspects like usability, appearance, and historical position. This comes into relation with what is being interacted with: a programming language and its notation, algorithmic processes, a synthesizer, and so on.

The custom of projecting the coding activity for the audience makes the projected interaction one of the primary aspects of the experience of live coding. Though live coding systems are quite personal,[83] interaction in live coding can be classified into two superficial categories based on this visual display: an orthodox style and idiosyncratic styles. Though not perfectly uniform, the orthodox style involves a text editor and an interpreter, and it can be observed in some live coding performances by McLean and Andrew Sorensen among others. Within this style, some may pay attention to the performer's choice of editor, syntax highlighting, presence or absence of visual cues upon code execution, and so on. Idiosyncratic styles may or may not involve the former, but they can include graphics, animation, or other interactive elements. Examples of idiosyncratic live coding interaction styles include some performances and systems by Griffiths,[84] Thor Magnusson,[85] and IOhannes Zmölnig.[86]

One example of the orthodox style of live coding interaction is that in McLean's perfor-

[83] Magnusson, "Confessions of a Live Coder."

[84] McLean et al., "Visualisation of Live Code."

[85] Magnusson, "The Threnoscope."

[86] Zmoelnig, "Pointillism."

mances using his Haskell library Tidal. In addition to its pattern representation and manipulation features, it allows the use of the interpreter for the Glasgow Haskell compiler (GHCi) and the Emacs text editor to live code patterns and trigger a synth using the Open Sound Control (OSC) protocol.[87]

An example of an idiosyncratic style of live coding is Griffiths' Scheme Bricks, which is a graphical environment for programming in the Scheme language which trades the editing of the signature parentheses of Scheme for manipulating colored blocks. It allows the user to graphically manipulate fragments of code in a way Griffiths feels differs from text editing as well as preventing coding mistakes like mismatching the number of parentheses.[88]

The nature of live coding when presented with a projection emphasizes an interaction with the audience in a way that other electronic music usually does not. McLean and Wiggins wonder about the feelings of the audience as a result of experiencing the projection, suggesting that some audience members may experience alienation even while others appreciate the opportunity to see the coder's interaction with the system.[89] While their anecdotal evidence says both are possible, that gathered by Andrew Brown suggests a positive reaction to be more common. However, he also notes that it can be perceived as showing off or a distraction.[90] The reason for this may be that some see the projection of coding as a flaunting of technical skill or a detraction from the performance due to the difficulty of under-

[87] McLean and Wiggins, "Tidal–Pattern Language for Live Coding of Music."

[88] McLean et al., "Visualisation of Live Code," p.3–4.

[89] McLean and Wiggins, "Texture," p.1.

[90] Brown, "Code Jamming," p.4.

standing the meaning of the code. This partly depends on the background of the viewer.[91] Considering their intentions is also important: audience purposes can range from dancing to deep consideration. For example, overemphasis of projected code might be a mismatch for an audience member wanting to dance, while it could be essential for those interested in learning how music can be represented in code.

Regardless of the superficial appearance of the live coding, many fundamental aspects are shared which have their own range of variation. For example, some performances may use previously written code, while others may be coded from scratch. Another example is the feedback from the interpreter, with some being sparse and others verbose.

Interaction with abstractions is one of those fundamental aspects of live coding. McLean makes clear one of the challenges of interaction in live coding is the higher level of abstraction for making sounds in live coding compared to manipulation of a traditional physical instrument.[92] The coder creates those many sounds by means of algorithmic processes. Brown characterizes those processes as "typically limited to probabilistic choices, structural processes and use of pre-established sound generators."[93] Those algorithmic processes are mapped to synthesizers. This creates some tension for the performer, who must juggle two somewhat dissimilar types of interaction: one with the algorithmic processes, and another with the synthesizer. In order to control these algorithmic processes, a user employees abstractions and the notation defined to express them in a given language. The level

[91] Bell, "Towards Useful Aesthetic Evaluations of Live Coding," p.5.

[92] McLean and Wiggins, "Tidal–Pattern Language for Live Coding of Music," p.1.

[93] Brown, "Code Jamming," p.1.

of abstraction among live coders can vary, and there is significant diversity in the notations used to express them, such as the parentheses-filled expressions employed by Sorensen and Griffiths, SuperCollider code used by Rohrhuber and others, and even differences in the Haskell used by McLean and this author. Naturally the idiosyncratic methods mentioned previously provide somewhat different means for interacting with the abstractions in their respective systems.

4.5 Rhythm in Live Coding

WordNet 3.0 provides a starting point with the following definition of rhythm: “an interval during which a recurring sequence of events occur”.

The Collaborative International Dictionary of English v.0.48 defines it as: “In the widest sense, a dividing into short portions by a regular succession of motions, impulses, sounds, accents, etc., producing an agreeable effect, as in music poetry, the dance, or the like.”

Toussaint cites at least 24 different definitions of rhythm from the ancient Greeks up to modern times.[94] The definitions run from Plato’s “an order of movement” to A.D. Patel’s definition as “the systematic patterning of sound in terms of timing, accent, and grouping”.

For the purposes of this document, awareness of this concept and the broad range it covers is sufficient. It is unnecessary to settle on a specific definition to progress with the arguments presented here.

Rhythm exists at many time scales, from the longest rhythms which use our long-term mem-

[94] Toussaint, **The Geometry of Musical Rhythm**, p.2–4.

ory to the fastest ones which we perceive as pitch.[95] Different cultures may perceive rhythm differently, such as an emphasis in Indian music on the cyclical nature of rhythm.[96]

A fundamental concept is that of the time interval between the start times of two events, or interonset interval (IOI).[97] Particularly in dance music, rhythm occurs in patterns which repeat once they have finished. Traditional western musical theory calls this a rhythmic ostinato.

An in-depth study of rhythm itself is beyond the scope of this document. Readers are referred to Sethares,[98] Cooper and Meyer[99], Clayton,[100] and Toussaint,[101] among others, for a more complete treatment of the subject.

A relatively complete list of live coding systems can be found here:

http://toplap.org/wiki/ToplapSystems

Of these, perhaps all have the basic facility for specifying an interonset interval.

SuperCollider possesses a library of functions for expressing patterns, which can become quite complex. These patterns are frequently used to describe sequences of interonset

[95] Sethares and Bañuelos, **Rhythm and Transforms**, p.1:1–9.

[96] Clayton, “Time in Indian Music,” p.15–16.

[97] Parncutt, “A Perceptual Model of Pulse Salience and Metrical Accent in Musical Rhythms.”

[98] Sethares and Bañuelos, **Rhythm and Transforms**.

[99] Cooper, **The Rhythmic Structure of Music**.

[100] Clayton, “Time in Indian Music.”

[101] Toussaint, **The Geometry of Musical Rhythm**.

intervals.[102]

The fundamental concept of Tidal is a cycle and its subdivisions. It offers abstractions for repetition, polyrhythm, the transformation of patterns, and the mapping of parameter changes over all members of a pattern. The abstractions for rhythm are mixed in with the abstractions for manipulation of the associated sampling synthesizer.[103]

Extempore exposes the temporal recursion in the editor along with a basic metronome facility and an abstraction for rhythmic meter. It offers the facility for specifying interonset intervals and manipulating them algorithmically, but it provides no additional abstractions explicitly intended for this purpose. It relies on traditional abstractions in Scheme and other functional languages for lists.[104]

A complete survey and comparison is made difficult by the rapidly expanding number of live coding systems, so it is left for future research.

4.6 Approaches to Evaluating Live Coding

Nick Collins has written several papers on the theory of live coding which touch on evaluation, including.[105] Alex McLean and Adrian Ward have written a satirical analysis of the

[102] McCartney, “SuperCollider.”

[103] McLean, “Tidal.”

[104] Swift, “Playing an Instrument (Part II).”

[105] Blackwell and Collins, “The Programming Language as a Musical Instrument,” Nilson, “Live Coding Practice., Collins, “Live Coding of Consequence..

stages of live coding.[106]

Meanwhile, despite a proliferation of systems and justifications, there has been little research on means for making judgments about live coding.

More concrete techniques for developing useful abstractions should be investigated, as well as methods, tools, or heuristics for judging abstractions, a need recognized in Green and Blackwell.[107]

[106] McLean, Ward, and others, “LivecodingGrades.”

[107] Green and Blackwell, **Cognitive Dimensions of Information Artefacts**, p.25.

5 My Works

This section describes the work that I have produced in this term of research. That work can be divided into three parts: music theory related to rhythm, development of the Conductive software library, and performances using that software. The music theory research describes methods for the generation of rhythm patterns, as well as explaining a method for increasing and decreasing the density of a given pattern and then moving between the various levels to create musically coherent rhythmic variation. The software was developed with basic abstractions for controlling real-time music. Abstractions were created to represent the rhythmic concepts described above. 19 public performances were given for audiences of between 10 and 350 people. All of the performances involved the Conductive software. The details of these three areas are described below.

5.1 Music Theory

The research on music theory involved concepts related to rhythm. This can be divided into two parts: the generation of rhythm patterns and then the variation of those patterns according to density. The research on the generation of rhythm patterns can be further subdivided into stochastic pattern generation and generation based on L-systems.

5.1.1 Rhythm Patterns

A rhythm pattern develops unity in part by including internal repetition. A rhythmic figure is a phrase of a particular length. However, within this phrase there are often components

which repeat or alternate. On this principal, a simple stochastic technique was developed for making patterns.

A phrase a specific length is generated. The length is chosen to satisfy the composer's musical intentions. In general, shorter phrases are easier to remember but can become stale sooner. A set of subphrases is also generated. The subphrases must be memorable for the listener to recognize them. To make them memorable, they should be short. In general, two or three IOI values is enough. Consider two sub-phrases in which the number 1 represents one beat: [0.25, 0.5] and [1,1,1]. In these two, the first subphrase consists of a sixteenth note followed by an eighth note. The second subphrase consists of three quarter notes. These phrases are short enough to be memorable. However, the subphrase [0.25, 0.5, 1, 0.25, 0.75, 1, 0.25] is too long. It does not contain repetition, and it may be difficult for a listener to easily recognize it the second time it is heard.

The particular values in the initial set of IOI values are based on a single unit. This unit generally provides the tatum unless its value is too small to be perceived as such (in which case some multiple of this unit is usually perceived as the tatum). As long as pattern length is divisible by the unit, a regular pulse is likely to be perceived. The IOI values in the subphrase are then multiples of the unit value.

This set also should not be too large. If too many different phrases are used, repetition of phrases is harder to recognize. The previous set is sufficient in some cases. Sometimes three or four is workable. However, a larger set of five or more is likely to strain the memory of most listeners. Consider the following set: [0.25, 0.5], [1, 1, 1], [1, 0.5], [0.25, 1, 0.25], [0.75, 1, 0.25]. This set is less useful for generating a memorable pattern.

A sequence of subphrases is selected from this set and placed end-to-end so that a longer phrase is formed. This phrase ideally displays repetition of elements to make the phrase memorable with enough variation for interest to be kept. For example, returning to the simple set above of [0.25, 0.5] and [1,1,1], we can call the first phrase A and the second phrase B. Now consider the sequence of ABBAB. This would expand to [0.25, 0.5, 1, 1, 1, 1, 1, 1, 0.25, 0.5, 1, 1, 1].

The phrase is then cropped to meet the length requirement. As an example, if three three-beat subphrases are selected for an eight-beat pattern, the last beat of the phrase is dropped. In the case described in the paragraph above, the values sum to 10.5. In the case of an eight-beat pattern, the last two items would be dropped and the final value would be truncated. That would leave the pattern of [0.25, 0.5, 1, 1, 1, 1, 1, 1, 0.25, 0.5, 0.5].

Additional details of this method for pattern generation and how it is implemented are described in the appendix on Conductive.

In addition to this stochastic technique, a method using L-systems was also developed.

Lindemayer systems, abbreviated to L-systems, were developed by Aristid Lindenmayer in 1968 as “a theory of growth models for filamentous organisms.”[108] They have since been used by many for the simulation of plant growth, for visual art, and to a lesser extent, music.

They are string-rewriting systems in which an input string, called an axiom, is transformed according to a set of rules in which each item in the string (a predecessor symbol) is rewritten as a successor string. By inputting this output back through the rule-set, successive

[108] Lindenmayer, “Mathematical Models for Cellular Interactions in Development I. Filaments with One-Sided Inputs,” p.1.

generations can be obtained.[109]

Here is a small example of a rule set, input, and seven generations:[110]

```
rules: a -> b
       b -> ab

input: a

output: b
        ab
        bab
        abbab
        bababbab
        abbabbababbab
        bababbababbabbababbab
```

L-systems which do not have one-to-one string replacement rules grow in length rapidly as seen above and require users to employ techniques to deal with that size.[111]

L-systems have been used in a variety of ways for algorithmic composition. Some examples from the literature include the following.

[109] DuBois, “Applications of Generative String-Substitution Systems in Computer Music,” p.9–11.

[110] Supper, “A Few Remarks on Algorithmic Composition,” p.50.

[111] DuBois, “Applications of Generative String-Substitution Systems in Computer Music,” p.13.

L-systems are frequently used for pitch content. Supper describes the use of L-systems and cellular automata for algorithmic composition.[112] Langston used L-systems to choose from previously composed musical phrases.[113] Morgan uses a system somewhat similar to Langston in which previously generated pattern fragments are chosen from and assembled according to an L-system-generated template.[114] One of the most complete and useful discussions of using L-systems for music is the dissertation by R. Luke Dubois, which describes various methods for generating patterns of pitches in monophonic melody lines and chords.

Worth and Stepney describe a set of L-system-selected rules by which note duration is progressively transformed.[115] Use of L-systems for duration or rhythm are described by Kaliakatos-Papakostas, Floros, et. al., including the use of what they call FL-systems, in which L-system output is constrained in length.[116] Kitani and Koike have also described a method of generating rhythms from L-systems in combination with a learning algorithm.[117] Liou, Wu, and Lee use L-systems to compute the complexity of rhythms.[118]

[112] Supper, "A Few Remarks on Algorithmic Composition," p.50–53.

[113] Langston, "Six Techniques for Algorithmic Music Composition," p.10–12.

[114] Morgan, "Transformation and Mapping of L-Systems Data in the Composition of a Large-Scale Instrumental Work."

[115] Worth and Stepney, "Growing Music."

[116] Kaliakatsos-Papakostas et al., "Genetic Evolution of L and FL-Systems for the Production of Rhythmic Sequences."

[117] Kitani and Koike, "Improvgenerator."

[118] Liou, Wu, and Lee, "Modeling Complexity in Musical Rhythm."

For more about L-systems, the dissertations of Dubois[119] and Manousakis[120] should be examined first. Readers are then advised to refer to the other items listed above.

A method for using L-systems for rhythm patterns was developed. First, an L-system with rules, an input, and a number of generations is specified. This yields a string. A set of potential IOI values is specified. From this set, a value is chosen randomly for each unique character in the string. Each character in the string is then exchanged for the IOI value to which it corresponds. Finally, the output is trimmed to a specified length.

It is convenient to use the previous L-system as the beginning of an example. Its fourth generation is “bababbab”. A set of possible IOI values could be [0.25, 0.5, 1]. Two of these chosen at random might be 1 and 0.25. These are assigned randomly to the unique characters in the string, which in this example are “a” and “b”. If 1 is assigned to “a” and 0.25 is assigned to “b”, then the resulting pattern is [0.25, 1, 0.25, 1, 0.25, 0.25, 1, 0.25]. If this is summed, it gives the value of 4.25. If it had been specified that the pattern length be 4, then the pattern would be trimmed to [0.25, 1, 0.25, 1, 0.25, 0.25, 1].

As will be discussed below, these rhythm pattern generators are expressed as functions, and the output is lists of numerical values rather than being written out in a traditional musical notation. In most cases, the output is simply written to the memory of the computer.

[119] DuBois, “Applications of Generative String-Substitution Systems in Computer Music.”

[120] Manousakis, “Musical L-Systems.”

5.1.2 Density

For musical development, the density of events in a rhythm should be variable. In other words, some periods of a rhythm pattern contain a higher ratio of events to length than others. Rhythmic density exists as a concept in classical Indian music called “lay,”[121] but no readily available algorithms for describing it were found.

To achieve this, the number of events in a pattern can be reduced to decrease rhythmic density or increased, thereby increasing density. At density levels close to that of the original pattern, the pattern remains recognizable. As density travels away from the original, the recognizability of the pattern diminishes.

The principle of increasing and decreasing the density of a pattern is simple, but there are an unlimited number of ways to generate the pattern variations at different densities.

A generalized method was developed to generate a list and a set of rhythmically similar lists of greater and lesser density and place them in a table indexed by level of density. The method for actually generating the variations can be swapped, meaning there is an infinite number of possibilities for generating those variations. Given a particular IOI pattern, a series of related patterns (both denser and less dense) is generated. It is built out to maximum and minimum density, making a list of IOI patterns ordered in terms of density. The lists are retrieved from the table according to a density value.

Some methods for producing those variations were considered. One stochastic method for doing so follows. When reducing density, an item from the pattern is chosen at random and

[121] Clayton, “Time in Indian Music.”

combined with a neighboring value to yield a similar pattern of reduced density. This process is repeated until the IOI pattern contains only a single item. When increasing density, an item is chosen at random and replaced with two items: an item of lesser value from the list of potential IOIs and the difference between the original IOI value and the lesser value. This is repeated until all of the items in the pattern are the smallest of the potential IOIs. By sandwiching the original IOI pattern between the less-dense and denser patterns, a table is generated.

A concrete example demonstrates the method:

- potential IOIs: 0.25, 0.5, 0.75, 1.0, 1.5
- input phrase: 0.25, 0.75, 0.5, 0.25, 0.75, 0.5, 2, 0.25, 2.75
- one level decrease in density: 0.25, 0.75, 0.75, 0.75, 0.5, 2, 0.25, 2.75
- second decrease in density: 0.25, 0.75, 1.5, 0.5, 2, 0.25, 2.75
- minimum density phrase: 8
- one level increase in density: 0.25, 0.5, 0.25, 0.5, 0.25, 0.75, 0.5, 2, 0.25, 2.75
- second increase in density: 0.25, 0.5, 0.25, 0.5, 0.25, 0.75, 0.5, 2, 0.25, 0.75, 2
- maximum density phrase: 0.25, 0.25

Based on a user-specified density value, a particular IOI pattern is chosen from the table. The user queries the table with a value between 0 and 1, and a linear conversion to a list index is done. The value returned is the IOI pattern at that index. Then correct next IOI value from that pattern is determined by the current beat.

This method increases the likelihood that two lists would be perceived as having a rhythmic relationship and decreases the chance that an audience would perceive a kind of discontinuous state change when switching from one to another (as might happen with some other possible techniques).

Another method which also preserves the rhythmic character of variations uses ratios to increase density. The user specifies a list of ratios, a lowest target value, and a limit to how small the IOI values in the rhythmic figure can be. A value is selected at random from the rhythmic figure. A ratio is chosen at random from the list of ratios provided by the user. The ratio is applied to the value, which is subtracted from the original value. These two new values are then shuffled and inserted into the rhythmic figure in place of the original value. The new rhythmic figure is stored in a list, and the process is repeated on this figure. This process is carried out recursively until all of the IOI values in the rhythmic figure are equal to or less than the user-specified lowest target value, producing a stack of increasingly dense rhythmic figures. The code for this procedure can be seen in the functions “densifier” and “densifier2”.

Consider an example in which a list, [1,1,1,1] is progressively densified according to a ratio of 0.5, with 0.25 being the lowest possible value. In this example, “it” is the ghci reference to the output of the previous command.

```
*> densifier 0.25 [0.5] [1,1,1,1]

[1.0,1.0,1.0,0.5,0.5]

*> densifier 0.25 [0.5] it

[0.5,0.5,1.0,1.0,0.5,0.5]
```

```
*> densifier 0.25 [0.5] it

[0.5,0.5,1.0,0.5,0.5,0.5,0.5]

*> densifier 0.25 [0.5] it

[0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5]

*> densifier 0.25 [0.5] it

[0.25,0.25,0.5,0.5,0.5,0.5,0.5,0.5,0.5]
```

This process continues until all of the values in the list are equal to 0.25, since it is the lowest possible value.

To compare this to traditional notation, it is as if a performer had a sheet of music with many lines, all of which referred to the same span of time. The performer would watch the conductor for the current beat and a line number, then play from the line corresponding to the conductor's signal.

5.2 Conductive

Conductive, a custom live coding system in the form of a library for the Haskell programming language,[122] was written to deal with the time constraints of live coding. It triggers a simple custom software sampler built with the SuperCollider synthesizer and loaded with thousands of audio samples. Through live coding, Bell manages multiple concurrent processes that spawn events, including the number of processes, the type of events spawned, and other parameters. A brief explanation of Conductive follows, with more details available in the

[122] Jones, **Haskell 98 Language and Libraries**.

appendix.

Conductive is a library for the Haskell programming language used for managing concurrent processes for real-time music. Concurrent musical processes are represented by a data structure called a Player. The Player refers to two functions: an action, which can be any function involved with system input and output and which it runs repeatedly; and an IOI function, which determines the IOI values that are interleaved between calls to the action function. Actions functions define what is done by a Player. These actions could include triggering a synthesis event or modifying the general system state. Currently, a sampler action is used predominantly. IOI functions define how long to wait between actions. Any methods available to the programmer could be used to generate those times, from returning a single value every time, to table lookup of values, to the calculation of values based on complex mathematical formulae. In addition to providing functions for managing those concurrent processes, it has some features for representing musical time and for algorithmic composition.

Some of those features for algorithmic composition include functions to describe and carry out the algorithms described in the previous section. One particularly important development in the library related to those algorithms is a generalized function for rhythm patterns and variations. To solve problems with lack of variation, the original function that handled density and base patterns was rewritten as a higher-order function, generalizing it to take external functions to handle those roles independently. A higher order function is “a function that takes a function as an argument or returns a function as a result…”[123] In this case it

[123] Hutton, **Programming in Haskell**, p.62.

takes two functions as parameters: one for the generation of the base pattern and a second for determining how the density of an input should be increased. The function is passed those two functions and a parameter determining how long the rhythmic figures should be. It uses the pattern function to produce a base pattern. It then processes the base pattern with the density function to create the complete table of a rhythm pattern and its variations.

The benefit of doing so is that the basic structure is already available before a performance and does not need to be coded at that time or recoded when the current method for generating patterns is longer useful. That means that generating patterns and then making a table of values from which can be read according to a density value can be accomplished more easily and in a greater variety of ways. John Hughes writes in his essay "Why Functional Programming Matters" that higher-order functions are one of two important kinds of "glue" that increase modularity, "the key to successful programming."[124] The use of higher-order functions in programming for aesthetic output has been described in.[125]

By changing the action functions, IOI functions, the data used by those functions, or the number of active Player processes, the live coding of a performance is accomplished.

5.3 Performances

Altogether, 20 public performances have taken place in 11 countries. All but one of the performances were either peer-reviewed concerts or performances by invitation. Many of

[124] Hughes, "Why Functional Programming Matters," p.1.

[125] McDermott et al., "Higher-Order Functions in Aesthetic EC Encodings."

them have been associated with academic conferences, while the others have been public concerts. Audiences have varied from academic researchers, audio engineers and programmers to punks living in a squatted grain factory. A complete list of performances can be found in the appendices in section 8.2. The performances explore how rhythms generated by the two methods described above sound at different density levels and focus on syncopated percussion sounds.

Performances were carried out using the Conductive system described above. In order to use this system, there are some prerequisites. The first of those is a Haskell programming environment. The Glasgow Haskell Compiler, which contains an interpreter (GHCi) that allows the interactive evaluation of source code,[126] is used by the performer to call functions from the Conductive library. The process of writing source code and sending it to GHCi is made more usable with vim (a text editor),[127] tmux (a terminal multiplexer)[128], and a vim plugin called tslime that allows text to be sent from the editor to the interpreter through tmux.[129]

As Conductive does not directly handle sound synthesis, a method for synthesizing sound is necessary. The scsynth component of the SuperCollider software suite is used to implement a sample player.[130] At present, synthesis events are programmed in Haskell and employ

[126] Jones et al., “The Glasgow Haskell Compiler.”

[127] Moolenaar, **The Vim Editor**.

[128] Marriott and others, “Tmux.”

[129] Coutinho, “Tslime.”

[130] McCartney, “SuperCollider.”

Rohan Drape's hsc3 Haskell library for communicating with scsynth.[131] The sampler uses samples that have been generally recorded and edited using Ardour,[132] and they have largely originated from hardware synths. All of the samples are individual sounds, from single-shot percussion sounds to bass samples. Most are wav files under 300 K.

Finally, in order to achieve a solid sound closer to that of commercial releases or broadcasts, the output of scsynth is processed through Calf plugins hosted by the Calf stand-alone host.[133] An EQ is followed by a multiband compressor and then a limiter, whose output is directed to the soundcard. Output is also directed to JAAA for monitoring.[134] Patchage is used for ease of routing.[135] Recording of performances is usually done with Ardour (in the case of audio). A screencast has been recorded with gtk-recordmydesktop (in the case of video).[136] Videos are often shot with a camera.

The venues ranged in size from a capacity of around 30 to one of 350. The types of venues included a squatted grain factory, garages turned into punk performances spaces, indie performance venues, bars, night clubs, art institute halls, museums, and sophisticated auditoriums. Projections ranged from the size of a very large monitor to movie-theater-sized projections. Sometimes the code was projected on top of my body and computer, at other times it was projected onto a large screen like one in a movie theater or on a nearby wall.

[131] Drape, **Haskell Supercollider, a Tutorial**.

[132] Davis, **Ardour**.

[133] Foltman et al., **Home @ Calf Studio Gear - Audio Plugins**.

[134] Adriaensen, **Kokkini Zita - Linux Audio**.

[135] Robillard, **Patchage**.

[136] Varouhakis and Nordholts, **recordMyDesktop Version 0.3. 7.3**.

That projection displays two terminals on my Linux computer: one terminal contains the Glasgow Haskell Compiler interpreter. The other terminal contains the vim text editor. The text is displayed in white, and it typically has used the Terminus font at a relatively small size. More recent performances have experimented with different fonts and larger sizes. Edits are visible as I make them in the editor. When I have completed code that I want to execute, I send it to the other terminal where the interpreter is running. The function appears in the terminal, along with any output the function is programmed to produce. Sometimes long blocks of output are printed in that terminal. The sound systems ranged from small PA systems to huge club sound systems which were very loud and projected deep body-shaking bass.

Audiences also varied widely. Some performances had young audiences of punks, artists, or programmers. Other audiences were considerably older and consisted mostly of academics. Some audiences were a mix of these and people of various ages and backgrounds. Audience sizes ranged from 10 to 350, with the average being around 40.

The music in all but two performances was in a percussive and bass-heavy style inspired by electronic dance music genres such as drum and bass and electro. The music employed thousands of percussion, bass, and noise samples to create an energetic stream of beats at either 140 or 160 beats per minute. They involved a library of samples for which I did all of the sound design. Such a style was chosen partly to suit the nature of the majority of the events, called algoraves. McLean and Collins describe algorave in the following way:

> Algorave is a current locus of activity where algorithms are explored in alliance with live electronic dance music; frequently they are the means of generating

> novel dance music on the spot from individual component events, or the manipulation of existing dance music segments. The nature of the algorithms for such production includes probabilistic generation within constrained parameters, and higher order transformation of pattern, and the interface of control varies from live coding to DJ-like instrumentation. The algorave website defines the movement by the statement 'sounds wholly or predominantly characterised by the emission of a succession of repetitive conditionals.' (http:// algorave.com), which seems to foreground repetition and conditional instructions, whilst underplaying random number generation… The format of an algorave is not clearly defined, and what goes on is ultimately the choice of the artists involved on the night, and audience members who choose to attend… certain features of the archetypal algorave are explicit: algorithms, music and dancing should be involved. Yet… audiences do not always dance… Algorave is not exclusively a preserve of live coders, but they have maintained a strong presence at every event thus far.[137]

The two performances that were not in such a style were the June 19, 2013 performance at the 9th ACM Conference on Creativity & Cognition 2013 in Sydney, Australia and the first performance on July 12, 2014 at Landbouwbelang in Maastricht, Netherlands. The former of these was a three part improvisation loosely based on classical Indian music and employed meters other than common time. The latter was a drone set as an opening for a set of performances by math rock bands. The second performance that evening was in the

[137] Collins and McLean, "Algorave," p.355–56.

algorave style described above.

The performance at the ACM conference was a 10-minute themed improvisation with three parts: an improvisation loosely but conceptually related to northern Indian musical tradition; an improvisation with percussion samples; and an improvisation that experiments with different subdivisions of time. In all three sections, the importance of rhythm and the perception of time was emphasized. Another focus of the performance is the density of events. Each section flowed from relatively sparse arrangements to extremely dense patterns of sound. The first section was an extension of a previous work, Ragalike, which was published in 2007 and was executed in the SuperCollider programming language.[138] It was inspired by improvised northern Indian music but used contemporary timbres. The second section involved transitioning between traditional percussion instrument samples and samples of electronic percussion instruments or otherwise synthesized percussion sounds. Movement in note density and rhythm patterns was sought. It was a further development of a video that appeared in the Computer Music Journal in 2011.[139] The final section explored the various ways that a beat can be subdivided. By dividing a single beat into increasing smaller fractions, different sonic qualities related to the sensation of the passage of time emerged. Various electronic timbres were put to use for this purpose.

In almost all of the performances, a general structure of trying to start from a simple drone and build intensity to a peak with breaks at proportional intervals was sought.

[138] Bell, **THE3RD2ND_011 - Renick Bell - Ragalike (Audio) (2007) [The3rd2nd.com – a Netlabel for Experimental Electronic Music]**.

[139] McLean, Magnusson, and Collins, “DVD Program Notes.”

6 Discussion

This section discusses the works above in light of the problem statement. First, a justification for making aesthetic judgments of live coding will be presented. With such a justification established, the aesthetic judgment of live coding will be examined from a pragmatic perspective. To do so, a brief introduction to philosophical pragmatism will be followed by a summary of the aesthetic theory of John Dewey. That theory will be revised to make it easier to apply, and then such an application will be carried out on two non-live-coding examples to make the theory familiar to readers. Finally, the theory is applied to live coding, both in general and to the specifics of this author's tools and performances.

6.1 A Pragmatic Aesthetic Theory

6.1.1 Justification for Considering Live Coding from the Viewpoint of Aesthetics

Aesthetic evaluation serves a variety of purposes. It can inform published criticism and provide one means of appreciating artwork. For some, a valuable use is its application to the creative process in which the creator is the first audience of the work. The creator can use the evaluation to adjust the work for maximum aesthetic effect. In the relatively new field of live coding, there is still a lot of opportunity for the improvement of aesthetic effect. This document particularly focuses on that goal, exploring John Dewey's Art as Experience[140] as a start for a system for useful aesthetic evaluations. The 20th century American philosopher John Dewey has exerted influence on the development of aesthetics by pursuing an intuitive

[140] Dewey, **Art as Experience**.

theory of art as experience. In addition, his writings on valuation can be applied to form a practical method for evaluation of such experiences.

First, however, a justification for considering programming to be an art should be examined. Harger acknowledges that artists are writing code as art.[141] Knuth says plainly that programming is an art, and he feels it is appropriate to use all of the connotations of art when describing programming. He describes his principal goal as a computer science educator as helping "people learn how to write beautiful programs…" He compares programming to composing music, and he describes an affective response to reading particular listings of program code. He finds the consideration of programming as art to beneficial to the field of programming.[142] Dijkstra also calls programming an art, saying "…the art of programming is the art of organizing complexity, of mastering multitude and avoiding its bastard chaos as effectively as possible."[143]

Fishwick says that programming is art.

> My central tenet is that it is both possible and desirable to craft software using aesthetic principles. The future goal of programming and computing lies at the crossroads of art and computer science. Achieving this goal requires laying down two important stepping stones which will carry us from program to artwork. The first is that we should craft programs as models. The second is

[141] Harger, "Why Contemporary Art Fails to Come to Grips with Digital. A Response to Claire Bishop. « Honor Harger."

[142] Knuth, "Computer Programming as an Art."

[143] Dijkstra, "E.W.Dijkstra Archive."

> that models should have aesthetic form. Combining both together yields the desired hypothesis programs should have aesthetic form. To the extent that aesthetic form requires a focus in art, the mechanisms, means and ways of art will become increasingly central to the programming discipline.[144]

In this description, a "model" can be seen as an abstraction.

Dexter et al. say that the source code of programs have both functional and aesthetic meanings. They also write of the intensity of experience possible in coding, echoing precisely the sentiment that John Dewey describes about the intensity of art experience.[145][146]

When we judge an abstraction, we are judging our experience of the abstraction. Kramer writes that "[the] level, benefit and value of a particular abstraction depend on its purpose."[147]

Arnheim agrees that the generalization carried out in abstraction points to its aesthetic nature.

> True generalization is the way by which the scientist perfects his concepts and the artist his images. It is an eminently unmechanical procedure, requiring not so much the zeal of the census-taker, the bookkeeper, or the sorting machine as the alertness and intelligence of a functioning mind.[148]

Programmers use aesthetics to judge abstractions. Shelley finds nothing strange in aes-

[144] Fishwick, "On the Aesthetics of Programming and Modeling."

[145] Dexter et al., "On the Embodied Aesthetics of Code."

[146] Dewey, **Art as Experience**.

[147] Kramer, "Is Abstraction the Key to Computing."

[148] Arnheim, **Visual Thinking**.

thetic evaluation of concepts.[149] Describing progressive stages of refining a program to calculate prime numbers, Pineiro concludes:

> Any programmer in the world will say that the programs get more and more beautiful. And what does he really mean? Well, he means exactly what he says, that he thinks that solution (4) is more beautiful than any of the other ones, that he likes it more, that it is more pleasing; what anyone means when he says that one thing is more beautiful than another. There is nothing strange, nothing peculiar in his usage of the word "beautiful", even if non-programmers (the majority of the human population) might not see any beauty at all.[150]

Kozbelt et al. did a quantitative survey on programmers' aesthetic experiences with code and found that both novices (84 percent of participants) and experts (100 percent of participants) find interacting with code to have an aesthetic dimension.[151] Aesthetics were used by both groups to make judgments about code, and such judgments were thought to be instrumental in software development.[152]

Knuth addresses the issue of aesthetic evaluation of code and finds it to be relative in at least some senses.[153] Pineiro states that judgments of beauty regarding a program may be relative depending on how the elements are grasped:

[149] Shelley, "The Concept of the Aesthetic."

[150] Piñeiro, "Instrumentality."

[151] Kozbelt et al., "The Aesthetics of Software Code," p.57–59.

[152] Ibid., p.61.

[153] Knuth, "Computer Programming as an Art."

> In the same way as the visual elements are what paintings are formed of, technical elements are what technical solutions are formed of. Which would be, in our programming example, the technical elements? I do not see any universally valid way of extracting the basic elements of a program, the ones listed below are only proposed as an example. Someone else might see some others, and therefore understand the program and its beauty differently. This is exactly the same thing that happens in the case of Mondrian's work, or of any other artist for that matter.[154]

Explaining further why such judgments may be relative, Pineiro writes:

> There is a long way to go if one wishes to generalise what I have said here to all sorts of instrumental actions, but I think that the relationships-method can be used in many other cases to explain why we consider some instrumental actions beautiful. In all cases we meet a similar problem: the observer does not know enough about the problem to be able to see the elements, and misses therefore the relationships and the beauty.[155]

Given that there is significant support for considering programs and source code aesthetically, it then seems sensible to ask whether there is an aesthetic framework which can be used to assist in such evaluations.

[154] Piñeiro, "Instrumentality."

[155] Ibid.

6.1.2 Dewey's Pragmatic Aesthetics

Cornel West, describing Dewey's approach, writes that: "… philosophy is neither a form of knowledge nor a means to acquire knowledge. Rather philosophy is a mode of cultural critical action that focuses on the ways and means by which human beings have, do, and can overcome obstacles, dispose of predicaments, and settle problematic situations."[156]

Leddy justifies interest in Dewey's theory: "It is a mark of the endurance and power of Dewey's aesthetic theory that it has been so frequently criticized and defended from so many different angles."[157]

Rorty writes that "On my view, James and Dewey were not only waiting at the end of the dialectical road which analytic philosophy travelled, but are waiting at the end of the road which, for example, Foucault and Deleuze are currently travelling."[158]

Cornel West believes that "… it is with Dewey that American pragmatism achieves intellectual maturity, historical scope, and political engagement."[159] He also called Dewey "the American Hegel **and** Marx! … the greatest of the American pragmatists"[160] West believes that "Dewey had far more immediate impact on society than Emerson, Peirce, or James."[161]

Dewey's aesthetics are defended by Shusterman, who acknowledges a creeping criticism

[156] West, **The American Evasion of Philosophy**, p.86.

[157] Leddy, "Deweyś Aesthetics."

[158] Rorty, **Consequences of Pragmatism**, p.6.

[159] West, **The American Evasion of Philosophy**, p.6.

[160] Ibid., p.69.

[161] Ibid., p.76.

in the middle of the 20th century:

> "My claim is that, since Dewey, Anglo-American theories of aesthetic experience have moved steadily from the former to the latter poles, resulting eventually in the concept's loss of power and interest. In other words, Dewey's essentially evaluative, phenomenological, and transformational notion of aesthetic experience has been gradually replaced by a purely descriptive, semantic one whose chief purpose is to explain and thus support the established demarcation of art from other human domains. Such changes generate tensions that make the concept suspicious… Dewey's prime use of aesthetic experience is aimed not at distinguishing art from the rest of life, but rather at"recovering the continuity of its esthetic experience with the normal processes of living," so that both art and life will be improved by their greater integration."[162]

This author believes that a revised version of Dewey's theory neatly captures all of the aspects of art and allows for rational explanation of different people coming to different conclusions about art. In other words, it shows how the relativity of judgments does not necessarily make those judgments irrational. Dewey's theory allows each experience to be judged by the standard appropriate to that experience, rather than forcing a single absolute and fixed measure or standard from which to judge all art, such as insisting on a universal definition of beauty and the need for all art to strive to achieve it. It also conveniently handles issues of time in art. By acknowledging that our experiences unfold over time in particular moments and accounting for that in our aesthetic judgments, their rationality is increased.

[162] Shusterman, "The End of Aesthetic Experience," p.33.

Here is a brief examination of Dewey's position that, though discarding much, should provide a foundation from which to move forward.

It is useful to consider the main points of "Art as Experience", an account of aesthetic experience as "essentially evaluative, phenomenological, and transformational."[163] Dewey begins by saying "the actual work of art is what the product does with and in experience."[164] Dewey calls art "a process of doing or making"[165] and an engagement with intention. "[Art] does not denote objects" in Dewey's theory.[166] An external product is a potential art experience depending on its audience. For Dewey it is preferable to think of art as an experience, and a painting or performance as a tool through which that experience can be realized.[167] This leads to a "triadic relation" in which the creator produces something for an audience which perceives it.[168] What the creator has produced creates a link between the creator and the audience, though sometimes the creator and the audience are the same, with the creator as first audience member. For Dewey, experience also means an interaction with an environment that is unavoidably human in every case and creates a feedback loop in which actions and reactions affect one another. These experiences are always composed of both physical and mental aspects. Experiencing the world means transforming it "through the

[163] Shusterman, **Performing Live**, p.21.

[164] Dewey, **Art as Experience**, p.1.

[165] Ibid., p.48.

[166] Leddy, "Deweyś Aesthetics."

[167] Dewey, **Art as Experience**, p.1, 113, 299.

[168] Ibid., p.110–11.

human context,"[169] and equally being transformed. A potentially infinite number of experiences can be derived from a single artifact or situation.[170] Because art is experience, it is always temporal in nature.[171] The aesthetic quality of the experience is emotional and involves both perception and appreciation.[172] "[Emotions] are qualities… of a complex experience that moves and changes", writes Dewey.[173] Dewey discards the distinction between what he calls "fine" and "useful" aspects of art, terms he saw applied to things like painting and industrial design respectively.

Sawyer finds this distinction between "the work of art", in which "work" can particularly be thought of in the sense of the verb or action, and the "art product" to be key.[174] The work itself is a composite or "total effect" of many factors.[175]

For Dewey, experience also means an interaction with an environment that is unavoidably human in every case and creates a feedback loop in which actions and reactions affect one another. These experiences are always composed of both physical and mental aspects. Experiencing the world means transforming it "through the human context", and equally being transformed.[176]

[169] Ibid., p.257.

[170] Leddy, "Deweyś Aesthetics."

[171] Ibid.

[172] Ibid.

[173] Dewey, **Art as Experience**, p.43.

[174] Sawyer, "Improvisation and the Creative Process," p.153.

[175] Dewey, **Art as Experience**, p.261.

[176] Ibid., p.257.

He argues for an atomic view of experience, in which the "mutual adaptation of the self and the object emerges" and concludes ideally with a feeling of harmony.[177]

Leddy explains that this leads to art being different on each perception of a creator's output, with a potentially infinite number of experiences. In this way, Dewey emphasizes the temporal nature of all art due to its origin in experience. Providing that experience with rhythm gives it aesthetic value.[178]

Dewey also argued that engaged craftsmen are artists, given their affection for the work. Leddy summarizes Dewey as rejecting the dualism of nature and spirit, favoring a connection between fine art and everyday life that has been lost in a contemporary society suffering from excessive categorization. Dewey seeks to do away with a distinction between "fine" and "useful" aspects of art, in which "fine" comes from the producer's experience in making it or the audience's experience engaging with it. Possessing utility is not a prohibitive factor in being "fine."[179]

For a more in-depth view of Dewey's aesthetics, Leddy's complete article is a good starting point.

A common criticism of Dewey noted by Leddy involves the atomic nature he assigns to experience. While at least in English a countable noun "experience" exists, arguments for experience having a discrete character fall apart upon close inspection as details and links to the past come into view. For example, the experience of a concert extends back to or at

[177] Ibid., p.45.

[178] Leddy, "Deweyś Aesthetics."

[179] Ibid.

least is colored by the expectations formed before the concert, the moment it was known that the concert would be held, the feelings already possessed about the performer, piece, venue, and so on. It also extends forward to looking at images from the concert or hearing recordings of it, reading reviews of it, etc. As those influences occur at a temporal distance from what may be assumed as the event and may occur simultaneously with or even part of other experiences, it becomes difficult to maintain a view of discrete experiences. It may be desirable to reject Dewey's atomic characterization of experience given the criticisms.

Deleuze and Guattari's rhizomatic structure is a concept through which the factors of an experience, including the art work, can be linked.[180]

6.1.3 Dewey and Valuation

Dewey wrote a considerable amount of material on valuation. His theory of valuation can be summarized as follows.

Value cannot be assigned in a disinterested manner.[181] Value is assigned to an experience according to the context of the experience (including but not limited to the culture it takes place in).[182]

Judgments are never "self-contained", but involve the weighting of factors to determine "what is to be done". Practical judgments inform appropriate courses of action and depend

[180] Deleuze and Guattari, **A Thousand Plateaus**, p.6–25.

[181] Dewey, "Theory of Valuation." p.17–19.

[182] Dewey, "The Logic of Judgments of Practise."

on the ends to be achieved.[183] Everything of value is instrumental in nature. Valuations themselves are instrumental for future valuations and action.[184] Every end is in turn a means for another intention in a continuous stream of experience.[185] Valuations are used to control the stream of an individual's experience.[186]

The value of something derives from how well it suits the achievement of an individual's intentions and the consequences of achieving those ends through those means. The object of an appraisal is also evaluated while considering its consequences with respect to other intentions held by the individual.[187]

Dewey would have evaluations of aesthetic experience used to enhance experiences through increased awareness of the causes and effects in the experience. In this way they are used to determine courses of action.[188] Recognition of a unity of means and ends in an experience is an important factor in the assignment of value. What enhances an experience is context-dependent and subject to revision as new knowledge is apprehended.[189] Value judgments are always in flux and susceptible to revision based on newly obtained experience[190] For Dewey, judgment is both creative in that it generates new ends and transformative in

[183] Ibid.

[184] Dewey, "Valuation and Experimental Knowledge," p.327–30.

[185] Ibid., p.329.

[186] Dewey, **Experience and Nature**, p. ix, 430.

[187] Dewey, "Theory of Valuation." p.52–53.

[188] Dewey, "The Logic of Judgments of Practise."

[189] Anderson, "Dewey's Moral Philosophy."

[190] Dewey, "Theory of Valuation." p.40–44; Dewey, **Experience and Nature**, p.1:399.

that judgments can cause reevaluation of what we have up to that moment prized. In this way, what is valued constantly evolves in an experimental way as increasingly informed judgments are made.[191]

6.1.4 A Revised Pragmatic Aesthetics

There are some problematic points to Dewey's theory, and as a result it has been criticized by or revised by various people, including Richard Shusterman[192] and John McCarthy and Peter Wright, the latter pair using it to explain interaction with technology.[193] This document presents a revision. It is divided into two phases: an experience phase and an evaluation phase. Before presenting those two phases, the concept of affect is discussed.

6.1.5 Affect

Affect is a complicated subject in itself, and the complexity is compounded by two different disciplines trying to comprehend it: philosophy and psychology.

Dewey describes the aesthetic quality of experience as emotional, but resists identifying discrete emotional states. Equating that emotional quality with the term "affect", it may be useful to adopt some reasoning on affects in Spinoza[194] and Deleuze and Guattari[195] to flesh

[191] Anderson, "Dewey's Moral Philosophy."

[192] Shusterman, **Performing Live**.

[193] Wright and McCarthy, **Technology as Experience**.

[194] Spinoza, **Ethics (Trans. RHM Elwes)**.

[195] Deleuze and Guattari, **A Thousand Plateaus**.

out the system. In describing Spinoza, Deleuze defines “affect” as a non-representational mode of thought distinct from ideas, which are representational. Perceptions are ideas, and a succession of ideas change one’s power of acting. Affect is not derived from the static comparison of ideas but is the temporal transition of focus from one idea to another leading to an increase or decrease in the individual’s power.[196] This temporal transition seems equivalent to Dewey’s “experience”. A particularly valuable example given in the lecture is the description of music. He describes music which he does not like as disrupting his network of concepts, understanding, and feeling, while music that he does like as resonating with that network. Such experience decreases or increases his power, respectively.[197] Saying that affective states vary according to a wide range of factors is different from saying that they are arbitrary, and such a statement is preferable to the tired statement that “all art is subjective”.

Seigworth and Gregg describe affect as:

> “the name we give to those forces – visceral forces beneath, alongside, or generally other than conscious knowing, vital forces insisting beyond emotion – that can serve to drive us toward movement, toward thought and extension, that can likewise suspend us (as if neutral) across a barely registering accretion of force-relations, or that can leave us overwhelmed by the world’s apparent intractability.”[198]

[196] Deleuze, “Lecture Transcripts on Spinoza’s Concept of Affect,” p.1–7.

[197] Ibid., p.21.

[198] Gregg and Seigworth, **The Affect Theory Reader**, p.1.

Massumi writes that “There seems to be a growing feeling within media, literary and art theory that affect is central to an understanding of our… culture… Affect is most often used loosely as a synonym for emotion.”[199]

According to Massumi,[200] Spinoza defines affect as the physical perception of something and the mental idea of that perception.

Gregg and Seigworth categorize theories of affect into eight groups, of which they assign the function of affect in the writing of Dewey as a means for understanding the connection between peripheral linguistic concepts and the senses,[201] but it seems that Dewey’s usage may also belong to other categories they describe. For example, the intersection of emotion and sensation seems to be one of Dewey’s targets.

Lawrence Grossberg borrows a phrase from Raymond Williams to describe affect as “the structure of feeling.”[202] He says “basically it’s become everything that is non-representational or non-semantic,[203] but rather than allow the category become so broad, a more specific meaning that fits with common usage is desired.

Massumi wants to distinguish between emotion, which he calls a quality, and affect, which he describes as an intensity in that essay. He gives it different rules from emotion.[204]

---

[199] Ibid., p.221.

[200] Massumi, “The Autonomy of Affect,” p.225.

[201] Gregg and Seigworth, **The Affect Theory Reader**, p.8.

[202] Ibid., p.310.

[203] Ibid., p.316.

[204] Ibid., p.221.

Rather than Massumi's complicated two-part representation, it is preferable to hold to the more commonly-used notion of affect, which WordNet 3.0 describes as "the conscious subjective aspect of feeling or emotion". "Affect" is preferable to "emotion" for the broader field that it includes; while a subtle feeling of comfort can be called affect, it is seldom designated as emotion.

Generally, while philosophy debates the exact meaning of the term considerably, the field of psychology has an accepted definition for affect which stays with the dictionary definition. That concept has been explained with particular clarity leading to a very applicable concept in Russell.[205]

According to Hilgard, Western psychology received the concept of affect from German psychology through the writings of William McDougall.[206] McDougall describes the cycle of mental activity as having three phases: cognition, conation, and affection. Cognition refers to the recognition of or thinking of some object. Conation refers to "the striving aspect of mental process", and he refers to the "passive aspect of experience" related to feeling as the affective phase.[207]

Russell describes confusion in contemporary scholarly work around the charateristics of "emotion" and its exact meaning[208] and gives an insightful example of different types of fear, from that experienced when watching a horror film to that of the extreme concern of an in-

[205] Russell, "Core Affect and the Psychological Construction of Emotion."

[206] Hilgard, "The Trilogy of Mind."

[207] McDougall, "Outline of Psychology." p.265–67.

[208] Russell, "Core Affect and the Psychological Construction of Emotion." p.145.

vestor about the market.[209] In responding to this situation, he proposes a system for describing affect which can give psychologists greater precision. It is based on axes of pleasure and intensity first described in,[210] yet Russell himself finds this insufficient and revises it in.[211] He explicity expands it to "Core Affect, Affective Quality, and Emotional Meta-experience."[212] In Russell's terminology, an art experience is possibly an emotional episode which, according to Russell, contains affect and responses to it.[213]

Russell's advanced conception of affect, called a dimensional model, is one good basis for this research to move forward on. Its main competitor is the categorical model[214] expounded by Jaak Panksepp and derived from the perspective of neuroscience.

Panksepp defines affect as "feelings that guide our thoughts and actions."[215] Rather than start with the Core Affect of Russell, Panksepp hypothesizes primary, secondary, and tertiary affect, with seven fundamental categories, which he capitalizes to distinguish them from folk-conceptions of emotions, as SEEKING, RAGE, FEAR, LUST, CARE, GRIEF/-PANIC, and PLAY.[216] This is derived from biological research involving the electrical and

[209] Ibid., p.146.

[210] Russell, "A Circumplex Model of Affect."

[211] Russell, "Core Affect and the Psychological Construction of Emotion." p.150.

[212] Russell, "From a Psychological Constructionist Perspective," p.85.

[213] Ibid., p.85.

[214] Zachar and Ellis, **Categorical Versus Dimensional Models of Affect**.

[215] Ibid., p.7:32.

[216] Ibid., p.7:33.

chemical stimulation of mammalian brains.[217] Basic affect corresponds to seven neurobiological systems, but cognitions do not appear to have "discrete neurochemical coding".

Panksepp does not agree that the bivalent system of Russell can account for the diversity of experienced affect.[218]

In Russell's scheme, what is called an affector in this research is called an Object ("object" with an uppercase "O"). For clarity of expression and accordance with standard English suffixes, affector is preferred.

It remains for further research to do a complete analysis of art experiences in terms of Russell's theory or Panksepp's theory. It is certainly beyond the abilities of this researcher at this point to choose one of the two theories.

Importantly corroborating Dewey's claims about valuation (discussed in a later section), Panksepp says "Affects, at their most primal level, reflect intrinsic valuative processes of the BrainMind."[219] Dewey says that Lotze assigns the emotions as "organs of value-judgments,"[220] and Dewey himself says that such affect can be "evidences of something beyond themselves… and so get a representative or cognitive status.."[221] Hookway finds the influence of Lotze and his ideas about psychology on another pragmatic philosopher, William James, in the idea of "'ideo-motor action', described by James as the principle

[217] Ibid., p.7:40.

[218] Ibid., p.7:41.

[219] Ibid., p.7:46.

[220] Dewey, "The Logic of Judgments of Practise."

[221] Ibid.

that ‘once an idea occupies the mind it will, unless obstructed, seek expression in action’.” Hookway finds this “link between concepts and habits of actions” to be very important for pragmatism.[222]

6.1.6 Experience Phase

An affect is an emotional state. An affectee is a person experiencing affects in an interaction with affectors. An affector is an object of perception that stimulates affects in an affectee. It can be something that exists physically or something abstract. A work of art is an affector which in some way was created, organized, or manipulated with the intention of it being an affector. A person involved with the creation or arrangement of an affector is an artist.

An art experience is the experience of affects in an affectee as the result of the affectee’s interaction with a set of affectors, with at least one of those affectors being a work of art. The art experience is the experience of those affectors either simultaneously or in sequence. Experience involves a possibly infinite number of affectors arrayed in a network structure in which they influence each other and the affectee either directly or indirectly. Changing the perceived network of affectors changes the nature of the experience.

6.1.7 Valuation Phase

The value of an affector is connected to the value of the art experience in which it is involved. The value of an art experience is determined by the affects experienced[223] and how

[222] Hookway, “Lotze and the Classical Pragmatists.”

[223] Bell, “Towards Useful Aesthetic Evaluations of Live Coding.”

well those affects and the other consequences of the experience and its affectors suit the intentions of the affectee.

Applying this aesthetic theory to an art experience means:

1. analyzing the intentions held by an affectee
2. determining the network of affectors that are present in the experience
3. examining the consequences of the interaction with those affectors, including resulting affects
4. determining the relationship between those consequences, affects, and the originally-held intentions
5. either (a) assigning value to the experience and its affectors according to those intentions or (b) changing intentions and then repeating this process with the new intentions

This process bears some similarity to the technique for analysis of the experience of technology described in Wright and McCarthy's "Technology as Experience."[224]

There are two ways to discuss valuation: how people assign value (a multiplicity of methods which may be discussable from a relatively objective viewpoint) and how people should assign value (a normative position). While this method may or may not describe how valuations are generally derived, this valuation heuristic is intended as a normative one that brings reason into conjunction with affect and will.

[224] Wright and McCarthy, **Technology as Experience**.

6.1.8 A Practical Application of the Pragmatic Aesthetic Heuristic: Mark Rothko, Untitled, 1955

An application of this theory to a practical example can make it easier to understand. For this purpose, the experience of an untitled Mark Rothko piece from 1955 can be used. This author attended an exhibition for the purpose of writing this document. The exhibition was titled “Guess what? Hardcore Contemporary Art’s Truly a World Treasure: Selected Works from the YAGEO Foundation Collection”. It ran from 2014.6.20-8.24 at the The National Museum of Modern Art, Tokyo.

The purpose of the exhibition was to contrast aesthetic value and market value of art works while making a statement about the necessity of private collectors in contrast to the museum collector. It also emphasized an intention of the collector, “living with art”.

My intention was to enjoy an afternoon of looking at art while gathering information for this document.

When approaching the museum, a poster of the museum was observed in the subway. Because I had not expected to see a Rothko on the way to the exhibition, it was a very pleasant surprise to learn that a Rothko would be on display. Immediately my intentions were altered from simply going to collect an experience to anticipating the experience of the Rothko. The painting could be found immediately to the left after entering the fifth section of the exhibition.

The 1955 work, predominantly orange and yellow, was called “Untitled”, with dimensions of 232.7 cm x 175.3 cm. The exhibition asked us to compare the painting to photographs of

the horizon at sea by a Japanese photographer, Hiroshi Sugimoto. While other works were exhibited on walls containing other pieces, this piece was on its wall alone.

The image itself was roughly the size that I expected, though I was surprised to find some fibrous material attached to the surface of the painting, like filaments from a brush or maybe the leg of an insect. There was also a black scuff mark on the frame on the upper right corner.

Some of the artist's brush strokes were more evident on close inspection. Because of the size of Rothko paintings, when photos of them are seen on the internet the details of the paintings are not apparent. Another aspect of Rothko works that web photos do not display well is the subtlety of color. The upper half of this particular painting has a greenish tint that is not visible in photos of the piece on the web. It had a very ethereal effect for me, throwing the whole painting off in its slight cooling versus the overall warm feeling of the painting. Being familiar with Rothko's style, the composition of the elements was very familiar and lent to the painting's aura of harmony.

The feeling from the painting was first surprise from seeing it in person. His paintings can and do suggest peacefulness and meditation, but there was something slightly dark from the tint in the upper half of the painting. His painting demonstrated a particularity of expression that made it stand apart from the other works, despite the curator's suggestion to compare it to Sugimoto's photographs.

When I was close to the painting, it was easy to observe the details. When I stepped away from the painting to take it in more generally, many people passed in front of me or sometimes stood directly in front of me. It was more crowded than usual in the museum because it

was a Sunday. While observing the painting, a woman stepped on my foot. Quite strangely, the same woman bumped into me or stepped on my foot a total of four times while I was in the exhibition, this time being the third and the fourth time being later in the exhibition. A bench was placed some meters away from the painting, but perpendicular to the painting, making it difficult to sit on the bench and enjoy the painting.

I was glad to have seen the painting, despite the distractions described above. I wish the curators had removed the scuff from the frame and attempted to find out if the fibres were part of the original presentation of the work or had somehow become attached in the painting's life on display. The experience stands as one of my favorite art experiences of recent memory.

The intentions I had in this experience can be listed out and include the following:

1. gathering information
2. enjoying art
3. seeing the Rothko painting

The network of affectors involved could include the things listed below:

1. exhibition poster
2. exhibition space
    1. wall
    2. bench
    3. other viewers
    4. woman who stepped on my foot
3. image

1. paint
2. strokes
3. color
    1. oranges
    2. greenish tint
    3. combinations
4. fibers
5. scuff
6. composition
7. planes
8. edges

Affects that were experienced included things like:

1. surprise
2. peacefulness
3. meditation
4. darkness
5. particularity
6. distinctiveness
7. ease of viewing
8. crowding
9. annoyance
10. distraction

11. pain
12. curiosity
13. gladness
14. regret
15. pleasure

Regarding achievement of my intentions, information could be gathered and the experience was more pleasant than I expected in some ways but slightly annoying in others. The pleasant surprise of being able to see a Rothko and the unexpected tint of the painting were enhancements of the experience. On the other hand, the blemishes and other extraneous elements on the painting and the woman stepping on my feet detracted from the experience by breaking my concentration on the image. I was most moved by the subtle composition and color combinations, the fuzzy lines of the edges of the various planes, and the interplay between these elements. Using this evaluation instrumentally, I resolved to try to go to the museum on a less crowded day. I also felt the need to check my own works carefully for blemishes so that they do not detract from my or others' experiences of them.

Interestingly, we can find Rothko's intentions here (quoted by Maria Popova from the book "Conversations with Artists"):

> "I'm interested only in expressing basic human emotions — tragedy, ecstasy, doom, and so on — and the fact that lots of people break down and cry when confronted with my pictures shows that I communicate those basic human emotions… The people who weep before my pictures are having the same religious experience I had when I painted them. And if you, as you say, are

moved only by their color relationships, then you miss the point!"[225]

http://www.brainpickings.org/2014/02/19/mark-rothko-on-art-selden-rodman/

While I enjoyed the experience of being absorbed in the visual environment the painting created and having some mild sensation of peacefulness or moodiness, I did not experience much of the basic human emotions or religious experience he was looking for. From his perspective, I may have failed to properly experience the work.

It is my argument that the artists' intentions can function as an affector in my experience, but they are not the final standard for judging my experience. Regardless of not having the experience that Rothko himself intended, I can still view the experience as positive and find instrumental value in it. Knowing his intention after the fact can cause me to alter my intentions with regards to the next Rothko painting that I am able to view. I am more likely to look for such an emotional experience from his paintings, though there is certainly no guarantee that I will have such an experience.

There was no need to otherwise revise my intentions after seeing the painting, other than an intensification of my desire to see a complete Rothko exhibition if given the chance. For the same reason, it further encouraged me to visit the Rothko Chapel in Houston, Texas. An instrumental judgment from the exhibition is a desire to achieve such subtlety and fascination through use of seemingly simple components in my own work. It is important to remember that instrumental judgments are always relative to the intentions of the affectee. While other people may benefit from those judgments, their true value exists in their ability to be applied to the intentions of the affectee. As a result, they carry personal and goal-oriented content.

[225] Rodman, **Conversations with Artists**.

A different viewer of this painting will have a different experience and different intentions, and therefore that affectee may come to differing value judgments about the affector and the experience.

6.1.9 Another Practical Application: Kankawa, Live at Pit Inn

I attended a concert on August 15, 2014 titled “Live KANKAWA-TAMAYA-RUIKE Super Electric Jazz Session (The Happening!!!)”. It featured the musicians KANKAWA on organ and Moog synthesizer, Tamaya Honda on drums, and Shinpei Ruike on trumpet and effects. It was held at the Pit Inn, in Shinjuku, Tokyo.

All of the seats have a small table in front of them. The stage area is blue. There were a few people smoking. The audience was of mixed age, young and old, and it was about two-thirds of capacity at around 30 or 40 people. Two large speakers were hanging from the ceiling on the left and right of the stage.

The first set started at roughly 8 pm. The trumpet player played solo, then the drummer entered and began playing. Finally Kankawa came out and joined them. The trumpet player was standing on the left. Kankawa was seated at the organ in the center, and the drummer was behind the drum set on the right. The trumpet player looked the youngest, with the drummer maybe five or 10 years older. They appeared half the age of the organist.

The trumpet player mostly played phrases no longer than two bars. Sometimes the trumpet player used a delay effect on the trumpet, and almost always there was a reverb effect. He played a fairly wide range of timbres. His left cheek blew up like Dizzie Gillespie, but only on the left side. There was an awkward feeling to the trumpet player’s playing.

The organist seemed to favor sounds like rock, blues, and a gospel-like organ style. He used the Moog sometimes, but the volume levels were often strange, sounding too loud and bursting out and above the ensemble sound. He set up song-like chord progressions, then evolved them roughly into new patterns, though sometimes with abrupt changes.

The drummer played rock or free rhythms, but did not swing much. There was little change in the density of his drumming, with either a soft or loud style and nothing in-between.

Their playing did not sound tightly synced. The organist and drummer performed multi-bar musical phrases, but the trumpet player was limited in this regard, seldom playing more than a two-measure phrase. They played a version of “House of the Rising Sun”, then their playing decayed to the end of the set.

The set ended, and a DJ started playing funk and rock to the right of the stage. The intermission felt quite long.

The trumpet and drum came out and played as a duo to start the second set, then Kankawa joined them. Soon they moved into a swing beat, and trumpet player played some multi-bar phrases. Their improvisation went back to something free. The trumpet player stopped playing and appeared to hesitate to come back in. When he returned, he was back to playing short bursts of sound that seldom span more than one measure.

Kankawa began playing blues and seems to enjoy himself, sometimes ignoring the others. Sometimes an atonal or blues ostinato emerged, and sometimes long bluesy chords. It felt like the staccato ostinati played by the organist were attempts to get the trumpet player to take solos, but the trumpet player always seemed to hesitate to enter, making awkward facial expressions.

There was a section where the organ was very delicate and the trumpet was making delicate noises too. This is followed by a brief section with the trumpet setting up a drone and an organ improv on top, but the drone did not continue very long.

Often the drummer or the trumpet player sat out while the other improvised with the organist. It was infrequent that all three played together. The trumpet player's sound was similar in ways to the bursts played by Miles Davis of on "Bitches Brew", but this may be simply because of the effects on the trumpet.

The volume crescendoed, and then they began playing "Amazing Grace". Kankawa gave signals at various times, and the drummer seems to follow the signals. The trumpet player, however, seemed not to pay attention to them. The drummer played the toms sometimes, but they were hard to hear over the organ. The playing went down in volume to an ostinato based on the "Amazing Grace" chord progression. Kankawa began to give a speech about recent political developments, such as in Iraq, Syria, Ferguson, the irony of President Obama being a black man, and so on. He then talked about the origins of jazz and the need for peace. He made an appeal that jazz is peaceful music and that war is wrong. He urged that it was about time for people to stand up (presumably to make some change in the difficult situations just described), especially the need for jazz people do something to prevent war. There was a reprise of "Amazing Grace" and the set ended.

The audience politely applauded for an encore, and Kankawa returned to talk about the kind audience and how jazz is the blues. However, he left the stage again without an encore performance. The audience began to leave. The time was then about 10:30 pm.

When evaluating this experience, first it is necessary to consider my intentions. I had in-

tended to enjoy a jazz performance and gather data for this document.

A large number of affectors are mentioned in the passage above, including:

- the venue, such as the speakers, stage, and furniture
- the audience, including their light conversations and smoking
- the line-up
- the instruments used by the players
- the playing styles of each member
- the parts that they play
- the styles of rhythms used, including the near-absence of swing in the drumming
- the density levels of their playing
- the various musical phrases
- the recognizable songs, “House of the Rising Sun” and “Amazing Grace”
- the DJ in the intermission
- Kankawa’s speech, including the words he spoke and his style of speaking

Some consequences of those affectors are mentioned, such as the organ or Moog drowning out the drums, the trumpet player ignoring the hand-signals of the organist, their scattered playing causing a perception of a lack of synchronization between the players.

Items like the last one belong to the set of affects resulting from the experience. I had feelings of dis-ease; a sensation of alack of group harmony; an awareness of hesitation, restraint, and blockage among the players; an feeling that they had an inability or refusal to groove; and so on. Some other affects included feeling the music was variously delicate, spirited, enjoyable, and nostalgic.

The negative affects overwhelmed the positive ones, mostly preventing affects of enjoyment from dominating the experience. In this regard, one of my intentions was thwarted. The lack of coordination between the players was disturbing, and the short unrelated bursts of the trumpet player were continuously disappointing and distracting. The drummer's lack of sensitivity towards the density of his playing and his lack of swing despite the jazz and blues idioms played by the organist prevented the session from feeling unified. It is possible that they did not intend to create a unified whole, but for me it just sounded chaotic and uninteresting.

On the other hand, the intention of gathering information for this document was a great success as the performance provided an excellent example to list out affectors and affects and examine their relationship.

Assigning value to the performance means drawing out instrumentally-useful judgments. Some of those judgments include the following. The performance made clear for me the need for performers in an improvising ensemble to somehow sound synchronized, so that there is some apparent logic in how the sounds are layered. It was also clear for me that soloists need ideas that extend beyond unrelated two-measure bursts that do not serve to illustrate a progression of ideas. Mixing idiomatic (blues and gospel) and non-idiomatic playing (free improvisation) probably has potential, but the players should make more efforts to either unify or more strongly contrast the styles they are playing. I will try to follow these judgments in future performances.

6.2 Affectors in Live Coding

Live coding consists of a network of affectors such as the musical output, including rhythms, timbres, harmony, rate of change, programming languages and libraries used, projection contents, interaction style, and performance space.[226] The network of affectors perceived by different affectees, such as the performer, music fans, programmers, or just the curious, and the affects they produce depend on each person's attention, knowledge, and experience. This section examines more carefully the affectors involved in the experience of live coding. It starts by investigating affectors in categories to which it is related, such as various musical genres and programming. It then looks in more detail at a central focus of this document: abstractions. After completing that examination, it discusses a key type of abstraction for live coding, generators, and then interaction with the various types of abstractions.

6.2.1 Affectors from Various Musical Genres

In order to understand the nature of live coding, it is valuable to consider the nature of music more generally. There is a large body of literature on the nature of music; consequently this section only mentions some of the many relevant points for consideration.

It seems that a useful system for evaluation would avoid bickering about genre and category and proceed more directly with investigations of value that yield helpful hints for producers. For example, one tar pit to avoid is distinguishing between "works" and "non-works". The revised pragmatic aesthetic theory presented earlier makes possible a broader and simpler

[226] Bell, "Towards Useful Aesthetic Evaluations of Live Coding."

analysis that should serve well for this purpose and is explored further below.

Following philosophical trends, the social aspects of music are increasingly recognized. Leppert describes music as a phenomenon with the potential for social subversion and listening to music as a path to knowledge.[227] Attali holds music to be a precursor of a new mode of economic production.[228]

However, while features like the social nature of music seem to invite agreement, there may be pessimism regarding finding agreement on other aspects of music. Predelli provides a detailed account of the debate about the nature of musical works and concludes with giving up on "unveiling interesting ontological features"[229].

It is still not clear how much of a common ontology is shared by different musical genres; Kania suggests they may differ extensively.[230] Without coming to a conclusion on this point, it will be valuable to examine some genres which are relevant to live coding: DJs, electronic music, jazz, and improvised music more generally. Jazz is of relevance to live coding because it shares the aspects of performance and improvisation. DJ music, particularly with the advent of software-based DJing, also shares potential for improvisation and a more technological bent. In addition, some live coding involves danceable music. Live coding fits within the larger category of electronic music and thus possesses characteristics belonging to the category.

[227] Leppert, "Music 'Pushed to the Edge of Existence' (Adorno, Listening, and the Question of Hope)."

[228] Attali, **Noise**, p.16:4–5.

[229] Predelli, "Has the Ontology of Music Rested on a Mistake?" p.18.

[230] Kania, "Pieces of Music," p.201–3.

Jazz can be said to at least sometimes consist of things like:

- the work being performed
- the improvised portions of the music
- the abstract contents of the improvisation (motifs, harmonic structures, rhythm patterns, and so on)
- the usage of stock riffs and other ready-mades[231]
- the composition of the band
- whether the instruments are amplified or not
- the level of inebriation of the audience

This catalog of components itself is contentious. Kania takes the position works do not exist in jazz.[232] The existence of works like Coltrane's Naima and Monk's Epistrophy (non-vocal compositions frequently performed by people other than the composers) would seem to show that not to be the case. Works exist in the account by Alperson.[233] Further, in Bailey "works" can be found along with "tunes" and "songs."[234] This issue will be set aside temporarily.

Examining the case of DJ performances can give us a partial catalog of factors in play, including:

- the recordings played

[231] Sawyer, "Improvisation and the Creative Process," p.157.

[232] Kania, "Pieces of Music," p.199–200.

[233] Alperson, "On Musical Improvisation," p.18–19.

[234] Bailey, **Improvisation**, p.48–49.

- how the records are mixed
- the use of vinyl and turntables, CDJs, or software-based mixing
- additional scratching done by the DJ
- filters and other effects applied to the DJ's mix
- the sound system the recordings are played on
- the DJ's physical gestures and sometimes voice
- the DJ's intention: to make the audience dance, display curatorial skills, create a new piece from existing recordings, and so on.
- the audience's intentions: to dance, to hear the recordings themselves, to hear the mixing, and so on.
- the amount of improvisation involved in the performance
- recorded DJ mixes versus live DJ performances
- the level of inebriation of the audience
- the venue, whether nightclub, basement, gallery, beach, or somewhere else

While Bell finds DJing to be largely about the relationship between what is live and what is recorded,[235] it is probably more likely that there is a rhizomatic linking among all of the factors (including but not limited to those above), as suggested by Deleuze and Guattari.[236] Such a view conforms to our common sense experience.

For example, the sound system affects the way the recordings are perceived, and particular recordings (for example, a strong ratio of bass to other frequencies) can influence the

[235] Bell, "Interrogating the Live," p.8.

[236] Deleuze and Guattari, **A Thousand Plateaus**.

response of speakers. Recordings which are ill-matched to an audience's intentions in turn influence the DJ to change selections or otherwise adjust a performance. This rhizomatic structure of relationships probably holds true for jazz and other genres as well.

That situation of the broader field of electronic music may differ from that of DJs, partly because the use of previously-recorded material is not a given. Electronic music in particular can be seen to have a wide distribution of creative agency across time, space, and people.[237] Paine and Drummond observe two approaches to what they term "computer-assisted music": control of pre-determined sound and the creation of sounds in real time. They acknowledge that these two approaches are sometimes undertaken simultaneously, and awareness of these two approaches is important for determining the authenticity of performances.[238]

Factors of electronic music, some held commonly by the genres above, include things like:

- hardware or software instruments
- studio recordings or performances in halls
- in the case of live music, the degree of liveness (tape music versus an improvisation with a PD patch, for example)
- the amount of improvisation involved in the performance
- the number of speakers (stereo works versus works for three or more channels)
- critical response to that work or related works

[237] Born, "On Musical Mediation," p.25–34.

[238] Paine and Drummond, "Developing an Ontology of New Interfaces for Realtime Electronic Music Performance," p.2–3.

This sampling serves to show that there are significant complexities and debate involved. Further investigation may be necessary.

6.2.2 Affectors from Programming

Programming can be thought of as the framing of a problem so that a computer and its attached peripherals can be used to solve the problem and then the writing of the instructions to allow it to do so. It is the manipulation of symbols representing abstractions to produce results.

Dijkstra writes that "…the art of programming is the art of organizing complexity, of mastering multitude and avoiding its bastard chaos as effectively as possible.[239]

Programming has been defined as using base elements, a means for combining them, and a means for abstraction, to deal with data and procedures.[240]

Having seen the difficulty of determining the nature of music, it can be anticipated that doing so for programming may be equally difficult. As with music above, a quick and naive survey might yield factors in addition to the ones mentioned above like:

- programming language choices
- the contents of the programming language
- the compiler used
- libraries

[239] Dijkstra, Dijkstra, and Dijkstra, **Notes on Structured Programming**, p.7.

[240] Abelson and Sussman, "Structure and Interpretation of Computer Programs," p.22.

- the underlying operating system
- the programming paradigm (imperative, functional, others)
- whether the program is compiled or interpreted

A software library is a collection of abstractions which are designed to be reused. Libraries are developed to save effort in the future. If a programmer wants to complete a particular task and a library contains an abstraction which can be used to complete that task, the programmer can simply use that abstraction rather than rewrite it. This saves the time and effort of the programmer.

There is debate about what a program even is.[241] The distributed agency in electronic music described by Born surely applies to programs as well.

Eden and Turner ask many other questions, such as whether programs are abstract or concrete, how modularity relates to abstraction and compositionality. For them, the relationship between source code and a running program remains an open issue, as does the nature of algorithms.[242]

Van Leeuwen provides a list of perspectives for philosophizing about informatics: information, computing, communication, cognition, design, and behavior. This list suggests the difficulty of developing a comprehensive philosophical account of programming.[243]

[241] Turner and Eden, “The Philosophy of Computer Science.”

[242] Ibid.

[243] Van Leeuwen, “Toward a Philosophy of Information and Computing Sciences,” p.24–25.

6.2.3 Affectors in a Live Coding Performance

Taking a survey of live coding and its components, a partial list of factors (many of which are shared by those fields above) includes:

- programming languages
- synthesis engines
- programming language libraries from others
- programming language libraries developed by the performer
- pre-performance code developed by the performer
- in-performance code developed by the performer
- projection contents
- editor
- interpreter
- sound output devices, like speaker systems or headphones
- performance space
- performer's actions in the space

Some explanation of these terms may be useful.

An editor, also know as a text editor or source code editor, is a computer program used for the writing and changing of text. In this case, the text is program source code. Source code is the text by which a computer program constructed. It is like the recipe from which an edible dish is prepared.

An interpreter is a computer program that reads source code passed to it by a user and executes that code immediately. This is in contrast to a compiler, which reads source code and creates another program from it. That new program is then executed at the discretion of the user.

When examining the code, there are various aspects such as:

- the syntax of the notation
- the symbols employed in the notation
- the volume of notation
- the abstractions and or algorithms that the notation expresses
- the coding style, such as formatting

Live coding does seem to point to new modes of production. The increasing popularity of live coding unrelated to music but done for education, mentioned by Gaspar and Langevin among others, seems to confirm this.[244] The distributed agency problem mentioned by Born exists for live coding. The existence of both solo and ensemble live coding is shared with both jazz and electronic music, and even tag-team DJing. Ready-mades in the form of pre-written code snippets exist. Applying external effects on top of the central content is a factor it shares with DJing, though it is uncertain if the aesthetic consequences are the same. It may or may not be useful to compare the use of code developed by others to sampling in hip-hop and other electronic music. Consideration could be given to how Van Leeuwen's list of perspectives applies to live coding.

[244] Gaspar and Langevin, "Restoring Coding with Intention in Introductory Programming Courses."

The issue of works seems to be a particularly thorny one. With regards to pre-performance preparation, the contents of performances, and audio or audio-visual documentation of such performances, live coding has a potentially contentious status when it comes to deciding if it has "works" that are interpreted in the manner of classical or jazz music or some other structure. The debate which Kania is engaged in would seem to become even more complicated here. For some,[245] instantiability appears to be key to calling something a "work". Kania further assigns importance to the creator's intention that something be reinstantiated at a future date as a criteria for calling something an art-work. From this viewpoint, some of the abstractions used in live coding could be seen as "works" by some, while other abstractions and the performances themselves are not to be identified as "works" (though at least Kania stresses that such a distinction should not be interpreted as lessening the value of performances). Compelling alternative viewpoints exist,[246] and one alternative view will be presented later in this dissertation.

6.2.4 Generative Processes as Affectors

Generative processes in an art experience have many aspects. In other words, they are compound affectors constituted of many affectors, including:

- the origin of underlying algorithm(s)
- the characteristics of the algorithm(s)

[245] Kania, "Pieces of Music," p.75–85; Brown, "'Feeling My Way'," p.115–16.

[246] Goehr, **The Imaginary Museum of Musical Works**; Davies, **Musical Works and Performances**; Born, "On Musical Mediation."

- a mapping of the algorithm to one or more synthesizers (audio in the case of live coding, but a visual or other synthesizer in other fields)
- the design of the internals of the process
- the implementation of the internals of the process, including its efficiency or elegance
- the design of the interface to the process
- the notation in code to express the process
- the degree to which the process has been abstracted and parameterized
- user interaction with the generative process
- the manner in which the generative process fits with other affectors involved in the experience

The necessity of considering all of these affectors in relation to the other affectors experienced follows from the theory above. This is supported by Cox, McLean, and Ward, who write that code should be evaluated both from its appearance as text and in the experience of it running.[247]

6.3 Experiencing the Affectors in Live Coding

Live coding is experienced in many ways at many levels due to the assemblage of affectors. That network of affectors is partially listed out in the section on the nature of live coding above. Each experience has its aesthetic aspect. The exact experience depends on the affectee, though. Consider these cases, with “fan” meaning a fan of live coding in general

[247] Cox, McLean, and Ward, “The Aesthetics of Generative Code.”

rather than of a specific performer.

type 1:	non-programmer	non-musician	non-fan
type 2:	non-programmer	non-musician	fan
type 3:	non-programmer	musician	non-fan
type 4:	non-programmer	musician	fan
type 5:	programmer	non-musician	non-fan
type 6:	programmer	non-musician	fan
type 7:	programmer	musician	non-fan
type 8:	programmer	musician	fan

Figure 1: a chart showing different types of affectees

Each row could be seen as a category of affectee, but even those who fall into the subset defined by that row can have radically different experiences even if they are assumed to encounter the roughly the same set of affectors.

As a first example, consider the row of “non-programmer, musician, fan”. Lack of programming knowledge may result in non-recognition of the control structures in use, how musical content is encoded, and even the mechanisms by which the performance is realized. On the other hand, they may not realize the poverty of control structures employed to realize the performance. Alternatively, the stimulation of trying to read code which one does not understand can create for some affects of novelty, mystery, and pleasure and alienation and frustration for others. Much of this depends on the live coding environment employed and any signals the performer gives to the audience, not to mention the previous experiences of affectees.

As a second example, consider the row of “programmer, musician, fan”. While a programmer viewing a live coding performance in an unfamiliar programming language has expe-

rience of programming in general to draw upon, illiteracy in the language used creates a different experience than if the affectee is familiar with the language used in the affector. A programmer can perceive programming elegance or understand the challenge of operations that are attempted, which is something that a non-programmer is more likely to miss. Perception of those things stimulates affects in programmer-musician-fans that cause them to assign value to the performance.

Some other considerations that could be made in light of the revised system include the following. A realization that different audiences will have different experiences and therefore different evaluations may mean a performer would like to make adjustments for particular audiences to maximize aesthetic impact. Focusing on the experience of using abstractions when programming may help to improve them. Alternatively, realizing the unwieldiness of particular programming methods (one node in the affector network) causes musical output to become overly static (another note in the network) may influence a practitioner to introduce other methods to add dynamism to a performance or change programming methods. As another example, if a particular affector is to come into play for an audience, the audience must be able to perceive it; The programmer may need to make special effort to make the existence of an affector perceivable for a particular audience.

6.3.1 The “Live” in Live Coding

One of the affects that can result from experiencing a live coding performance is liveness. As the method of live coding bears “live” in its name, this affect is essential. There has been considerable debate about what liveness means, what brings it about and what the

consequences of feeling it are. While music performed by traditional methods in front of an audience rarely encounters controversy as to whether it is live or not, computer music frequently faces skepticism.

Those who perform music on laptops or attend such performances often hear discussion about whether a performance is live or not. The fact of attending a performance is somehow not enough, as the audience often doubts whether the performer is actually performing the music or doing some unrelated task. Even when they trust that the performer is only engaged in music, they still suspect that the performer is exerting very little real-time influence on how the music proceeds. This affect, liveness, needs more examination to understand what the affect actually means and what can trigger it.

This author, along with Oliver Bown and Adam Parkinson, conducted a survey in which listeners were asked about their perceptions of liveness and performer control in laptop performances. The people who took the survey listened to several short segments of live electronic music performed in part or in whole on laptops. They were asked questions about those segments. The questions asked what the listeners thought were the means by which the sounds were performed and how "live" the segments sounded.[248]

The study suggested that audiences which perceive performer activity consequently feel the affect of liveness. That activity can be actual gestures or audible gestures, and it is associated with improvisation. Audiences can recognize the presence of generative processes, but those processes do not impede the sensation of liveness.[249]

[248] Bown, Bell, and Parkinson, "Examining the Perception of Liveness and Activity in Laptop Music."

[249] Ibid.

The analysis of the survey indicates that liveness may have three types. The first is a kind of bodily or physical liveness through perceptible physical action. Another is cognitive liveness in which the performer's mental processes are perceived or determined through some means to be affecting the course of the performance. Finally there is a type of liveness which just expresses the fact that the performer is present in front of the audience and deemed to be presenting a work regardless of whether their presence appears to affect the course of the performance.

The resulting analysis can be compared to the conception of liveness according to Auslander.[250] He claims that there is nothing essential in the mode of presentation of musical material or the material itself that distinguishes live from non-live. Auslander describes liveness from a cultural-historical perspective, arising as a cultural concept in response to the emergence of recorded media. He puts liveness in opposition to the concept of mediatised. Auslander says "the anxiety of critics who champion live performance is understandable, given the way our cultural economy privileges the mediated and marginalizes the live."[251]

The paper distills four points argued by Auslander:[252]

1. Liveness and mediatization derive their difference from historical factors rather than ontological conditions.[253] Live performance cannot remain

[250] Auslander, "Liveness, Mediation and Intermedial Performance"; Auslander, **Liveness**.

[251] Auslander, **Liveness**, p.42.

[252] Bown, Bell, and Parkinson, "Examining the Perception of Liveness and Activity in Laptop Music," p.18.

[253] Auslander, **Liveness**, p.7–8.

"ontologically pristine."[254] "Mediatized" means "a product of the mass media or media technology"..[255]

2. "live and mediatized performances are parallel forms."[256]
3. The live gets overwhelmed by the mediatized.[257]
4. Liveness exists dually in opposition to the quality of being recorded,[258] yet he doesn't see it as existing before 20th century audio-visual technology because he does not see written and oral accounts as recordings.[259]

The paper then presents the following counter-arguments to Auslander's points:

1. Liveness is a perception based on the prior perception of performer activity or decision-making.
2. Liveness and mediatization can co-occur. Live laptop music involves the performance of the mediatized. Mediatization may in fact amplify perceptions of liveness.[260]

The paper then describes why liveness is a result of perceptible performer activity:

> The first is implied from our responses. Further, all of the performances are

[254] Ibid., p.40.

[255] Ibid., p.5.

[256] Auslander, "Liveness, Mediation and Intermedial Performance," p.7.

[257] Ibid., p.9; Auslander, **Liveness**, p.37.

[258] Auslander, **Liveness**, p.51–52.

[259] Ibid., p.52.

[260] Bown, Bell, and Parkinson, "Examining the Perception of Liveness and Activity in Laptop Music," p.18.

> highly mediatized, being made of layers of material that is not instrumentally performed. Audiences call something 'live' when they feel aware of performer decisions and physical activity in the moment of the performance, and this is independent of the media used, whereas for Auslander 'live' comes as a historically determined duality with 'mediatized'.
>
> But while 'live', in this usage, is said to appear in 1934, seemingly a time of increasing use of media technology, its use is said to mean 'in person.'[261] 'In person', with the meaning of 'by bodily presence' dates to the 1560s.[262] Where 'live' is satisfied by the concept of 'bodily presence', a live performance is one in which the bodily presence of the performers can be perceived, with apparently increasing levels of presence. That bodily presence may have become more abstract and cognitive in response to the laptop mode of performance, crossing over into the technological sphere.

It follows that if an affectee is aware of the performer's activity, the affect of liveness is more likely to occur. In the case that the affectee is seeking to experience that liveness, this affect can contribute to a positive evaluation. In the case that liveness might be attended by performance errors, unpredictable elements, or other inevitabilities of something happening in real time, liveness might be undesirable. If an abstraction is able to positively impact perception of performer activity, the performance may seem more live as a result of the abstraction. The next section deals with the experience of abstractions.

[261] Harper, "Live."

[262] Harper, "In Person."

6.3.2 Experiencing Abstractions

Abstractions and their features are affectors in the network, either directly perceived or indirectly felt as a result of their influence on other affectors.

Live coding, or writing or editing a programming while it is running, poses many difficulties to its practitioners. Some of these difficulties come from a usability perspective, such as program text comprehension and dual-task interference,[263] and abstractions are a central part of this.

The difficulty in using abstractions can impair the other affectors. They can be opaque, hiding their contents from people lacking particular knowledge which would otherwise permit recognition of the true role of the abstraction in the experience. Other affectors can impact whether and to what degree abstractions are perceived. Affectors which impact that perception include the size and clarity of the projection, the type of sounds that are heard simultaneously, whether the text in the editor is scrolling or not, and so on.

The problem of difficulty of understanding abstractions in live coding is recognized by Aaron, Blackwell et. al. who attempted to achieve “an abstraction gradient” as some abstractions “discourage casual use” . They additionally report that well-chosen and specific higher-level abstractions served the project well.[264]

Blackwell, Green, and Nunn say that abstractions “are potentially hard to understand. Moreover, choosing the appropriate abstractions can itself be a problem (premature commit-

[263] Nishino, “Cognitive Issues in Computer Music Programming.”

[264] Aaron et al., “A Principled Approach to Developing New Languages for Live Coding,” p.382.

ment); changing them can difficult (viscosity); and changing one abstraction can lead to unforeseen changes to other ones (hidden dependencies)."[265]

Further difficulties described in Green and Blackwell include potential for errors in creation, extended creation time, and effort of maintenance.[266]

Green and Blackwell find that "[for] exploratory tasks, they are problematic. Highly skilled designers can use them, but they are most likely to be exploring the abstractions needed as well as exploring for a solution given the abstractions."[267]

It seems that much live coding is done by such designers or those seeking such skills. As such, it is felt that while efforts should be made to decrease their difficulty of use, their use has the potential to enable operations which are otherwise unachievable. Further, it is precisely the challenges and rewards of exploratory activity with abstractions that provides some of the aesthetic value of live coding for some affectees.

Brooks further warns that "…at some point the elaboration of a high-level language becomes a burden that increases, not reduces, the intellectual task of the user who rarely uses the esoteric constructs."[268]

This points to the commonly referred to "Law of Leaky Abstractions" coined by Spolsky: "All non-trivial abstractions, to some degree, are leaky." Spolsky explains the trade-off, saying "[all abstractions leak], and the only way to deal with the leaks competently is to learn about

[265] Blackwell, Green, and Nunn, "Cognitive Dimensions and Musical Notation Systems," p.3.

[266] Green and Blackwell, **Cognitive Dimensions of Information Artefacts**.

[267] Ibid., p.25.

[268] Jr, **The Mythical Man-Month**, p.186–87.

how the abstractions work and what they are abstracting… the abstractions save us time working, but they don't save us time learning." The systems for which the abstractions provide an interface encounter problems, forcing the user to deal with the underlying system rather than the abstracted interface, with one example given by Spolsky being functions for TCP networking.[269]

The leakiness of abstractions for coders using them is greatly amplified for affectees other than the performer: coders unfamiliar with the abstraction are more likely to struggle to understand them, and that is increased to a much greater extent for non-coders.

Simonyi writes that since abstractions are expressed in programming languages, users hesitate to adopt changed languages for economic reasons. In addition, the notation of programming languages and their textual nature sometimes prevent adoption. Finally, he laments that advanced abstractions can be domain-specific, preventing their adoption by other domains.[270] The inherent limitations posed by the representative nature of abstractions in programming for music is acknowledged in chapter 4 of Magnusson.[271]

Blackwell and Collins compare the usability of programming languages for live music with the interfaces of commercial music software and explain the challenges that users of ChucK face.[272] These challenges can likely be attributed to other live coding environments. Of

[269] Joel Spolsky, "The Law of Leaky Abstractions."

[270] Simonyi, "Intentional Programming," p.1.

[271] Magnusson, "Epistemic Tools."

[272] Blackwell and Collins, "The Programming Language as a Musical Instrument," p.3–6.

particular interest here is the characterization of ChucK as diffuse.[273] Additional abstractions are given by Blackwell as the solution to this problem.[274]

When implemented, abstractions are accompanied by notation. Green says that a "system = notation + environment". Revising Green, it could be thought that abstractions are the link between the notation and the environment. The notation makes the abstractions representing the environment employable. Conceiving of the right abstractions may simplify the notation needed to accomplish a programming goal and or counter the deficiencies of the environment. Used with a model of behavior, the system requirements can be stated and differing dimensions may emerge. This makes trade-offs clearer. Alternatively, it may be possible to modify the notation to counter the deficiencies of the environment, and vice versa.[275]

Achieving proper role-expressiveness seems to rely heavily on the notation accompanying the abstraction: choosing the proper names, including characteristics such as morphology, length, and grammatical sensibility, for the abstractions and their parameters is important.[276] For example, a function can be named "raveBassDrop" instead of "rd1" to inform the audience of an intended musical effect. A performer intending to be cryptic might do the reverse, like in Zmoelnig's "Pointillism."[277] On the other hand, "rd1" is much shorter and therefore faster to write. With typing speed being an issue in live coding, Brown and Sorensen sought

[273] Ibid., p.6.

[274] Green and Blackwell, **Cognitive Dimensions of Information Artefacts**, p.43.

[275] Green, "Cognitive Dimensions of Notations," p.2.

[276] Liblit, Begel, and Sweetser, "Cognitive Perspectives on the Role of Naming in Computer Programs."

[277] Zmoelnig, "Pointillism."

compact code constructs.[278] Given sufficiently high-level abstractions, the speed of typing should become much less of an issue.

For another simple example, consider a function producing a synthesis event requiring two parameters. Notation for the underlying abstraction (the same in both cases) can vary wildly. Each function call has its own advantages and disadvantages:

```
myFMSynthesizer (SynthesizerEventData { amplitude = 1.0, modulation = 0.5})
```

```
syn 1.0 0.5
```

The elegance with which an abstraction solves its target problem can make the abstraction an affector in an art experience. For example, a live coder trying to stimulate an audience of coders might write functions live which exploit recursion, currying, or other techniques to elegantly express musical content, and do so from scratch to exhibit virtuosity for affectees who are capable of understanding them.

Increasing the awareness of abstractions must be a factor in the depth of an experience of live coding. For some audiences, a performance could survive aesthetically on compelling use of advanced abstractions, while for other audiences the audio output would trump any consideration of the abstractions employed. It is up to the performer to adjust for those audiences accordingly. The intention of the performer then becomes very important. If the goal is clarity of expression of the mechanisms of the performance for the audience, then the abstraction and its notation need particular care. If the intention is simply to show

[278] Sorensen and Brown, “Aa-Cell in Practice,” p.3.

the audience the activity of the performer through a projection (or even to perform with no projection at all), the abstraction just needs to suit the needs of the performer for expressing the desired output. It is not clear that these two are complementary, and cases where they may be in conflict can be imagined. In the “syn” example above, the performer can type “syn” and two numeric parameters much faster than the longer function name and the notation for the data structure parameter. What is an advantage for the performer’s speed is a disadvantage for the audience’s understanding.

Each individual abstraction, once perceived, is taken in a different way depending on the person. One major factor is the affectee’s knowledge. An example is the use of >> in coding done by this author. First, someone who is not familiar with the symbol may just ignore it. However, some may guess. A non-programmer might interpret it as an arrow, pointing a direction. It might give an impression of movement. Another might think of it like a quoting symbol in an email. A programmer with Unix experience might initially think it is a redirect, as in: ls >> files.txt. Another might see it repeated several times in a line and wonder if it is a pipe. A Haskell programmer will know it has a type signature like this:

```
(>>) :: forall a b. m a -> m b -> m b
```

The documentation describes it as used to “[sequentially] compose two actions, discarding any value produced by the first, like sequencing operators (such as the semicolon) in imperative languages.”[279] Three types of people leads to three different experiences of a single symbol, pointing at the leaky nature of abstractions.

[279] Various, **Haskell Hierarchical Libraries**.

6.4 Evaluating Live Coding from a Pragmatic Viewpoint

Collins describes one direction for the aesthetic evaluation of live coding:

> The more profound the live coding, the more a performer must confront the running algorithm, and the more significant the intervention in the works, the deeper the coding act. In programming, small changes can have grand repercussions; it is not impossible to imagine changing a single character or connection to achieve substantial consequences (for instance, substituting one data array for another via similar names at a key juncture, moving from "for" to "fork" or "and" to "rand," making a single rewiring between waiting complex processes).[280]

It may be interesting to put this starting point through the revised "art as experience" system described above. "The profound", "significance", and "deepness" are affects, and a performer is an artist. "A running algorithm", an "intervention", "small changes", "a single character", "substantial consequence", "data arrays", functions, "rewirings", and processes and their complexity all are affectors joined in a network. Whether the audience perceives those things and how they perceive them determines whether they experience those or other affects.

[280] Collins, "Live Coding of Consequence," p.209.

6.4.1 Intentions in Live Coding

Live coders have expressed a broad and diverse set of intentions,[281] though the central and common intention is the real-time creation and presentation of digital content,[282] often audio content, and doing so particularly through use of algorithms.[283] Some seek the challenge and the chance to improvise,[284] while others desire flexibility of expression.[285] There are live coders who want to code collaboratively.[286] It is the aim of some to communicate algorithmic content to an audience,[287] making the coder's deliberations clear,[288] as well as showing how that activity is guided by the human operator.[289] Some aim to demonstrate virtuosity,[290] interact more deeply with a computer,[291] or discover new musical structures.[292] This can mean

[281] Magnusson, "Confessions of a Live Coder."

[282] Brown, "Code Jamming," p.1.

[283] Brown and Sorensen, "Interacting with Generative Music Through Live Coding"; Thielemann, "Live Music Programming in Haskell," p.1.

[284] Collins et al., "Live Coding in Laptop Performance," p.1–2.

[285] Blackwell and Collins, "The Programming Language as a Musical Instrument," p.3; Magnusson, "Confessions of a Live Coder," p.2.

[286] Brown and Bischoff, "Indigenous to the Net"; Sorensen, "Impromptu," p.4–5; Thielemann, "Live Music Programming in Haskell," p.1.

[287] Brown, "Code Jamming," p.3.

[288] Sorensen, "Impromptu," p.1; McLean et al., "Visualisation of Live Code," p.1.

[289] Brown and Sorensen, "Interacting with Generative Music Through Live Coding," p.9–10.

[290] Sorensen, "Impromptu," p.1.

[291] Collins et al., "Live Coding in Laptop Performance," p.322; Brown and Sorensen, "Interacting with Generative Music Through Live Coding," p.4.

[292] Sorensen, "Impromptu," p.5.

trying to describe generative processes efficiently,[293] either in terms of computational power necessary or code necessary to express an idea. The intention can even be ironic and in opposition to the goal of clear communication of algorithmic processes for the audience.[294]

6.4.2 Applying the Heuristic to Live Coding

Following these steps, the heuristic described above can be specifically applied to the experience of live coding.

1. determine the intentions held by the live coder, such as making an algorithm clear or making sounds that fit in a particular genre
2. determine the network of affectors, including libraries, programming language, other tools, projection
3. determine the consequences of interaction with those affectors, such as ease of use, confusion, mystery, and so on
4. determine the relationship between those consequences, affects, and intentions, such as a library that does not allow the particular musical expression desired or an algorithm that produces a particularly desirable affect
5. assign value to the experience and the affectors or change intentions, such as accepting a strange algorithm because of its surprising result

[293] Brown and Sorensen, “Interacting with Generative Music Through Live Coding,” p.2.

[294] Zmoelnig, “Pointillism.”

This process bears some similarity to usability heuristics such as those suggested by Nielsen.[295] While this and similar writing has influenced the heuristic presented in this document, there are also distinct differences. For example, Nielsen describes 10 heuristics to be used generally and pointing towards a single goal of ease-of-use. The heuristic described here should be similarly universally applicable but points toward a different goal or set of goals depending on the evaluator (affectee). Nielsen's heuristic evaluation is intended for use by software designers, while the proposed heuristic is for a more general aesthetic evaluation and for use by any person involved in an experience (in other words, any affectee). Nielsen's heuristic requires several evaluators. This heuristic has a somewhat different purpose and should be effective with even a single evaluator producing an evaluation valid for that person alone.

The difference in purpose is important. Usability studies are conducted to make software easier to use. This is a possible but not essential component of the purpose of this heuristic, which is to draw instrumental value from an experience. A simple example would be that a designer would make modifications to software if multiple evaluations showed that it was difficult to use. Using this heuristic, a user of that same software may decide to choose a different piece of software to complete the task. Alternatively, the user may decide that the challenge is fun and get more deeply involved in the use of the software.

In addition, usability studies often begin from the assumption that software under testing already produces the correct result. The tests are conducted to make sure that the users can manipulate the software to achieve that result as easily as possible. When applying this heuristic to live coding, in many cases it could be said that live coders are trying to confirm

[295] Nielsen, "10 Usability Heuristics for User Interface Design."

whether the software produces the correct result or not. In this regard, a live coder can use the heuristic to change the software to get a more correct result. Usability, meaning ease-of-use, is one goal, but it is not the only goal.

It is beyond the scope of this research to examine in further detail the theoretical relationships between the consequences of an experience of live coding including its affects, the intentions of the affectee, and the valuation assigned to that experience and its affectors. Such analysis is reserved as a future target of research.

6.4.3 Evaluating Abstractions

Kramer writes that the "level, benefit and value of a particular abstraction depend on its purpose."[296] Said another way, abstractions have to be chosen carefully with the goal firmly in mind. A continual refinement of those abstractions toward an evolving ideal is generally desired. A live coder choses purposes for the performance and then tries to achieve abstractions as close to ideal for those purposes as possible, revising both the abstractions and the purposes as a result of experiencing this experimental process. Though their value is contextual, abstractions can be judged generally insofar as those general judgments prove useful. Given that, some general conclusions about abstractions follow.

Two targets are making higher-level abstractions so that the intended musical output is easier to achieve and designing abstractions and their notation so that the intention of the programmer regarding the relationship between the abstractions and the affectees is achieved.

[296] Kramer, "Is Abstraction the Key to Computing."

Despite the potential drawbacks, higher-level abstractions can significantly improve the performer's ability to manipulate the generation of musical material. Well-chosen and appropriate abstractions can make expressing musical content easier[297] or even make possible what would otherwise not be. That improves both the experience for the performer and the audience.

Good abstractions make it easier to do things appropriate to those abstractions;[298][299] if the abstractions are low-level, doing complex things at a higher level requires verbose code.[300] It follows that the usability of an environment is linked to the nature of the available abstractions.[301] For example, when an abstraction which aggregates a series of steps is used in place of manually carrying out the steps, action slips can be prevented.[302]

Abstractions must be chosen with care. The notation of abstractions should be chosen with usability in mind. Compatibility with existing systems can reduce the fear of adoption and increase the chances that others outside the target domain find the abstractions useful and contribute to their development.

Targets for higher-level abstractions could include things such as briefly-expressed constructs for managing state or methods for managing time at a levels including sub-beat, phrase, and full composition.

---

[297] Aaron et al., "A Principled Approach to Developing New Languages for Live Coding," p.3.

[298] Krueger, "Software Reuse," p.134–36.

[299] Green and Blackwell, **Cognitive Dimensions of Information Artefacts**, p.24.

[300] Jr, **The Mythical Man-Month**, p.224.

[301] Green, "Cognitive Dimensions of Notations," p.1–2.

[302] Norman, "Categorization of Action Slips," p.10.

Shaw suggests identifying the essential properties of subsystems and how they interact and targeting these things for the development of higher-level abstractions.[303] She says this will help to deal with the complexity of such systems.[304]

Developers seeking to provide general-purpose tools need to make efforts to make abstractions as high-level as possible while simultaneously keeping them as general as possible. The need for high-level abstractions becomes more severe in the case of live coding, where the value of concise expression is amplified due to its being a live performance. On the other hand, Brooks warns against the creation of excessively arcane abstractions.[305] Following these guidelines and others described in this paper, it is believed that the usability and potential of a live coding system can be increased.

Simonyi further explains that abstractions and their corresponding operations can simplify the finding of a solution when those abstractions are taken from the non-programming abstractions of the problem domain and match the expectations of users on how those abstractions are used together.[306]

The goal is what Simonyi calls “intentional programs”:

> From the programmer’s point of view, intentions are what would remain of a program once the accidental details, as well as the notational encoding (that is the syntax) had been factored out. Intentions express the programmer’s

[303] Shaw, “Larger Scale Systems Require Higher-Level Abstractions,” p.1.

[304] Shaw, “Toward Higher-Level Abstractions for Software Systems,” p.119–20.

[305] Jr, **The Mythical Man-Month**, p.22, 224–25.

[306] Simonyi, “Intentional Programming,” p.1–3.

> contribution and nothing more.[307]

For example, the design of the Haskell programming language intends to free the user from the heavy tasks of managing resources and sequencing operations to focus on the important details of the algorithms being implemented. This was seen by Sylvan and others as needed to escape from the loop-constructing emphasis of imperative languages and the consequent noise that accompanied it. Being able to put consideration of those matters aside was seen to increase flexibility and ease of use.[308]

Following Simonyi's recommendation, choosing preexisting abstractions from the musical domain as targets is a good method to reduce the difficulty of the mental operations of the programmer when using them. Returning to the list given by Shutt and extrapolating, it can be asked what kind of higher-level data types and control structures would benefit programming for music.

Berg gives the suggestions of "reduced instruction set composition" and "evocative simplifications of compositional activity". In describing Xenakis, he mentions shapes, areas, densities, and contours. Koenig is described as expressing compositional rules and tendencies. Berg also suggests "behavior that changes over time."[309]

Other targets could include things such as briefly-expressed constructs for managing state or methods for managing time at a levels including sub-beat, phrase, and full composition.

[307] Ibid., p.3.

[308] Sylvan, Amir, and Wahler, "Why Haskell Matters - HaskellWiki," p.

[309] Berg, "Abstracting the Future."

This reflects Shaw's suggestion to develop abstractions at a system-organization level.[310]

Wishart provides a strong account of such potential targets. He describes characteristics of sound and form, many of which are based on utterance and natural phenomenon. The role of the perception of sound and achieving freedom from Western notation are highlighted.[311] He refers to "lattice sonics" (sounds using organizational methods related to grids) and those that fall outside of grid-based expression.[312] Mixing the two categories, some particular aspects sampled at random and listed alphabetically include articulation, counterpoint, density, distribution in space, dynamism, gesture, grain, landscape, masking, mass, modulation, morphology, nesting, permutation, space nature, stability, timbre, time-based transformation methods, topology, types, and so on.[313]

It can be argued that the flexibility of such abstractions should be kept in mind by library developers so that developed abstractions do not unduly color the resulting compositions. They should be general enough that a number of genres could employ them, and according to Brooks' warning, their frequency of use should be potentially high to avoid being esoteric. However, another programmer may desire very unique abstractions that would somehow give performances a distinct flavor. The intentions of the programmer can be seen as critical.

[310] Shaw, "Larger Scale Systems Require Higher-Level Abstractions," p.1.

[311] Wishart, **On Sonic Art**, p.12:326.

[312] Ibid., p.12:7–8.

[313] Ibid.

6.4.4 Experiencing and Evaluating Generators

McCormack, Bown et. al. present a list of ten questions about generative art. One of those questions asks what characterises good generative art.[314] A dialogue providing answers to their questions through use of the theory above shows how it can be applied and might achieve the "more critical understanding of generative art" they say is needed. A selection from their questions and this author's responses follow, with comments related to live coding added.

> Why is generative art in need of special quality criteria?[315]

Proper consideration of the role of the generative process in the experience is needed. Because a generative process is a compound affector, consideration of all of its components is also necessary. It is also necessary to ask to what degree the affectee is aware of the generative affector and its components.

> Is it better considered alongside other current practices?[316]

Experiences exist in relation to one another. Past experiences influence present ones. In addition, generative elements appear alongside non-generative elements in every case, and the two have influence on one another. For example, in live coding sometimes audio samples are triggered as a result of a generative process. While the triggering is the result of the generative process, the audio may result from sound design efforts that may or may

[314] McCormack et al., "Ten Questions Concerning Generative Computer Art."

[315] Ibid., p.9.

[316] Ibid., p.9.

not rely on generative techniques. In the case that they do not, the practice of creating generative processes is being considered in conjunction with the practice of sound design. Thinking critically about the relationship between these two seems useful.

> Consider two important properties that differentiate generative art from other practices. The first is that the primary artistic intent in generative art is expressed in the generative process. This process is what the artist creates, and as such should arguably be the subject of scrutiny in appreciation of what it produces.[317]

This seems unnecessary. First, it may be difficult to define a "primary artistic intent" in some cases. Artists frequently possess a variety of intentions.[318] It may be hard in many cases to choose just one as primary, and it is not certain that doing so is necessary or makes the work any better. For example, eating a meal at a fancy restaurant serves a practical intention of satisfying hunger, and it may also work towards various aesthetic intentions, such as enjoying exquisite flavors, appreciating an environment, engaging in stimulating conversation with friends, and so on.

Further, it seems conceivable that a generative technique might be used as a means towards an end that the artist gives higher priority. For example, consider a live coder whose primary artistic intent is making an audience dance. In this case, the intent to use generative processes is subservient to the primary intention of stimulating and maintaining a full and energetic dance floor.

[317] Ibid., p.9.

[318] Bell, "Considering Interaction in Live Coding Through a Pragmatic Aesthetic Theory."

Using a generative process is just one tool that an artist has for producing a work, along with a collection of other tools. An artist should have the freedom to select appropriate tools in every circumstance. It does seem appropriate, however, that the generative process and its output figure in the evaluation of the work as affector and consequently in the evaluation of the total experience.

> Secondly, the way this process is interpreted or realised is also the locus of artistic intent, and is intimately intertwined with the first property. The basis of all generative art resides in its engagement with process. So the locus of artistic intent should include the motivations, design and realisation of the process…[319]

While the generative process may or may not be the center of the artist's activity, this point recognizes that the generative process is actually a compound of several factors. Each factor plays a role in evaluation of the generative process and an experience of it.

> Put simply, the "generative" and "art" parts are inseparable. Process in generative art should be considered the primary medium of creative expression, implying that the exclusive or predominant use of creative software or processes designed by others in one's generative practice is problematic.[320]

Calling the use of tools from others problematic is too strong. The total experience should be judged. For example, the use of a standard algorithm but mapped in an original way should

[319] McCormack et al., "Ten Questions Concerning Generative Computer Art," p.9.

[320] Ibid., p.9.

still be able to cause affects of admiration of originality, surprise or novelty, or other positive affects. There seems to be no reason that employment of a generative process designed originally by someone else could not be used by another artist. It can be thought of as jazz sax players playing saxes that someone else has manufactured, or singers making use of songs from composers other than the singers themselves. The fact that a painter has not manufactured the paint in her painting is rarely a reason to evaluate the experience of the painting poorly.

> Understanding an algorithm's subtlety or originality opens a fuller appreciation of the eloquence of a generative work. But this is a significant problem for most audiences, reinforced by focussing on the surface aesthetics of the art object as is often seen in computational generative art, where the computational process is rarely directly perceptible.[321]

Collins also notes this problem.[322] Any art experience is taken differently by different affectees. Knowledge of an area closely related to an affector changes the experience, but it is too much to ask that every affectee have working knowledge of all aspects of each affector. It is better to accept the knowledge that an affectee brings to the experience and allow that background to give them an authentic experience, even if it is a different experience from an affectee who is an expert. An artist can reveal the generative aspects of an art work, and that transparency can be good as long as it is in harmony with the intentions of the affectee (such as the artist). Providing enough information so that the audience can understand the

[321] Ibid., p.10.

[322] Collins, "Generative Music and Laptop Performance," p.69.

generative process could be an intention of the artist, but it does not seem to be necessary. Collins suggests good program notes to increase the functioning of generative processes as an affector for audiences.[323]

Games are one example of an artwork that has generative aspects which are not the main focus of the piece and in which the generative aspects only function indirectly in the experience of affectees. Live coding works similarly for affectees with relatively less knowledge of the means of live coding but possessing intentions such as immersion in electronic music or dancing.

Brown and Sorensen provide a detailed account of their experience with generators in live coding in.[324] Some discussion of those points follows.

> As the example above shows, the way in which an algorithm is represented can impact upon its utility for the live coder.[325]

This certainly seems true. The artist experiences that representation directly, and other affectees in the audience it may experience it directly if that representation appears in the projection or experience it indirectly through its influence on the output sound or projection contents.

> The description length and complexity of an algorithm plays a large factor in its appropriateness for live coding. Algorithms such as neural networks, evolutionary algorithms, agent based systems, and analytic systems are all

[323] Ibid., p.68.

[324] Brown and Sorensen, “Interacting with Generative Music Through Live Coding.”

[325] Ibid., p.6.

> affected by issues of description complexity. The longer the description of the algorithm, the more time will pass writing the code in which the programmer is unable to pay attention to other aspects of a performance.[326]

It may then be advisable to use the generators in a generalized sense, meaning already abstracted and available as functions, and code around the parameterized aspects of the generator. The ability to code a generator from scratch could be one intention of a live coder, making this an important point. However, other intentions can make the approach of using an abstracted generator as a library function very effective. While the way that a process is implemented can be factor, it is not always the case that the implementation is the most pertinent affector for an affectee. In many experiences it can be almost invisible.

> When programmers make a decision to abstract code away into a library, an abstract entity which can only be accessed as a 'black-box', the ramification is that they no longer have the ability to directly manipulate the algorithmic description.[327]

Parameterization might mitigate this problem. A higher-level function can take functions as parameters, in which case some structure is fixed but other structure can be controlled by the programmer in a live setting. The general framework can be coded in advance, leaving a key component to be coded in a performance or to be selected from a body of pre-coded components. It also depends on how the library is accessed, since in some cases that code may still be malleable.

[326] Ibid., p.7.

[327] Ibid., p.7.

The flexibility of abstractions and code in this way appears as an affector for affectees with programming knowledge, and the resulting affects again are determined partially by intention.

> Many grammars, pattern matching and analysis systems require a substantial amount of look-ahead for decision making and also often require the generation and scheduling of material into the medium to distant future. We have found these types of algorithms to be not very valuable in practice as they limit our ability to affectively respond to other concurrent processes, input devices and, most importantly, fellow performers.[328]

It seems that some of these processes could be run as if they were non-real-time, that is given their targets in advance and allowed to generate their data silently. Once their output has been generated, the performer can select from or edit it. That data can used by another process. Naturally it would be less responsive to real-time interaction, but it may still be useful. Still, the opinion that Brown and Sorensen express reflects their unique experience of those algorithms. Others may find that such limits are impulses to other creative activity, leading to positive affects.

> we have identified a set of algorithms that we have found particularly valuable… [including] probability, linear and higher order polynomials, periodic functions and modular arithmetic, set and graph theory, and recursion and iteration.[329]

[328] Ibid., p.8.

[329] Ibid., p.8.

Here, Brown and Sorensen describe their positive experience with various techniques. It is worth noting that some affectees may lack necessary awareness that such techniques are being used for these affectors to work directly, though their results are certainly felt indirectly.

> Many simple processes, such as repetition, can become tedious, while others, such as randomness, can seem featureless and uninteresting… This balancing of control and surprise is a constant challenge for generative sound artists and our experience suggests that at present it is better handled by the performer than by some computational 'agent.'[330]

Brown and Sorensen further describe their experience with various types of generators. It does seem that others might feel differently. Artworks which have been valued highly by many in the past have relied heavily on both repetition (Alvin Lucier's "I am sitting in a room.", for example) and randomness (such as Marcel Duchamp's "Three Standard Stoppages"). This shows the variability of experience, partly due to the framing of that experience and differing intention.

Brown and Sorensen write that "generative processes should be:"

- "succinct and quick to type"
- "widely applicable to a variety of musical circumstances"
- "computationally efficient allowing real-time evaluation"
- "responsive and adaptive by minimising future commitments"

[330] Ibid., p.10.

- “modifiable through the exposure of appropriate parameters”[331]

Brown and Sorensen derive actionable criteria from evaluating their experience, which seems productive.

The matter of being succint is up to the notation representing the generative process, rather than particularly being a factor of the generative process itself. Naturally, the larger the number of parameters that that abstraction requires can influence the amount of typing, but even one abstraction with a given number of parameters can be represented notationally in various ways, some of which are more concise than others. This relates directly to their last item, which is unavoidably in conflict with the first point. However, their point is correct in that frequently good abstractions are parameterized well to achieve maximum generality.

This leads to their second point. The width of application is also up to the performer. A generative process which is specific though frequently used may not require application to other cases in order to be evaluated positively. Proper parameterization will increase generality, but some abstractions may still remain fairly specific in their use-cases, which may still not be a factor which would cause a negative evaluation.

Modularity can reduce the commitment required to a generative process. If the process generates data that is in turn read and rendered by another process, then the user simply has to change what the rendering process refers to for its data. This could refer to the commitment required in the generation of the data itself. The need to change the direction of a particular generative process can be reduced if the processes are flexible enough to be discarded freely and instantiated in abundance, or so on. It is likely, though, that only the

[331] Ibid., p.1.

performer is likely to directly experience these affectors.

It is important to remember that generators can lead to a change in our intentions. Though a generative process may be conceived with a particular goal in mind, after becoming familiar with its output, intentions can change. What may have be imagined as a simple and boring test may be later considered to have interesting output, while an implementation of a generator that perfectly expresses the intention it was begun with may turn out not to have interesting output, causing a change of intentions.

6.4.5 Experiencing and Evaluating a Live Coding Experience: "A Study in Keith" by Andrew Sorensen

This piece can be viewed in several places on the internet, including Wikipedia at this URL:

http://en.wikipedia.org/wiki/File:Study_in_keith.ogv

By going to the this page, the familiar Wikipedia layout is seen. The file downloads fairly quickly on my optical fiber connection, and I am soon able to watch the video. I play the file using a media player program and the video window appears.

The piece starts with no sound. Sorensen writes some comments (his name and email address) and then starts to write code. His code has syntax highlighting, meaning that different parts of the code are colored according to what the code means. The colors are mostly bright and warm colors of orange and purple. First he defines a piano synth and loads its parameters. Then he sets a metronome which will provide the pulse for his performance. It is set to 110 bpm. He then clears the screen of code. The code which he is writing is

Scheme, and therefore it is made up of S-expressions, which are visually recognizable as code delimited in parentheses.

New code begins to appear on the screen defining chords. He pastes in code snippets so that the code appears very quickly. He then begins to edit the code defining the function "chords". Finally he executes the function and some sound begins, which is a very slow progression of chords played by a piano synth. In my case, those sounds emanate from small speakers on a stand next to me in my office.

He proceeds to edit the definition of "chords" and execute it to get changes in the output, slowing increasing the rate of the chords. Every time he executes a function, a section of code is highlighted and blinks briefly. At other times the lavender highlighting helps him by identifying which block of code he is editing. By editing the timing section of the function, the rhythm of the chords becomes more variable. He then edits the progressions and starts a bass section of the piano. Then he starts to edit a new function, "pulse", and when he executes it a pulsing piano stab begins to accompany the chords being played. He then adds another new function, "runs", which is defined to play notes according to the aeolian mode. As he is editing it, the pulse and chords are playing. The effect is very much like a noodling piano improvisation reminiscent of the sound of the artist in the title of the piece, Keith Jarrett.

He stops editing "runs" and goes back to edit "pulse", changing the nature of the pulse. He then returns to the "chords" function and changes the timing of the chords as well as adding a call to the "runs" function. When this executed, sometimes the chords are interspersed with runs of mostly rising piano notes. The effect is still very much that of an improvising

pianist. The pulse gives the improvisation grounding and movement, and the chords and runs seem to be building in intensity.

He continues to edit “chords” and the other functions, changing parameters of the now complete code structure. The pulse disappears, and the piece suddenly becomes quieter as he strips out notes. The bass has gone completely. The rhythm is changed to become sparse. Finally, he stops the performance and writes in a comment at the bottom of the screen “the end!”.

Abstractions that are used include functions for setting up audio, the “*metro*” function for controlling IOI values, functions which iterate over lists, various music-related abstractions such as “pc:diatonic” and “play-note”, and some math functions like “cos”, and “random” which chooses random values from a list). However, the piece is essentially defined in these three functions: “chords”, “pulse”, and “runs”.

The piece lasts 11 minutes and 38 seconds. On my computer, the video window disappears when the video has stopped playing, leaving silence and my regular computer interface in its place.

My intention in watching this video was to learn something about live coding, enjoy again this video which I have seen many times, and to write an explanation of it for this document.

The affectors are listed above.

Affects of watching this include the following. I feel nostalgia, since I have watched this video many times over the years, and it reminds me of Keith Jarrett as well. It feels somewhat mechanical compared to Keith Jarrett, but I continue to appreciate its musicality, particularly after hearing so many other live coding performances. Over the years, though, I experience

decreasing amounts of pleasure hearing melodic music. The aeolian mode piano which was very sweet to my ears years ago now somewhat irritates me. His speed of editing show his proficiency with his system and evoke feelings of admiration.

Deriving some instrumental judgments from the experience, I resolve to improve my own proficiency with my system to match Sorensen's. He pastes in code blocks, but some of this feels like it could be abstracted into higher-level abstractions. In particular, if regularly coding a piano, it seems that "chords", "pulse", and "runs" are things that a performer might want to run in every performance. As such, having readily at hand abstractions for these things might free up some of the programmer's mental resources and allow the programmer to concentrate on details such as shaping the performance through melody, dynamics, or rhythm. The lesson would be to identify similar functions which are often written, generalize them, and then place those generalized versions in a library where they could be easily accessed in a live coding session. Regardless, I am inspired by what certainly feels like a seminal video and hope I can make such an inspiring video myself someday.

6.5 Experiencing and Evaluating My Live Coding Performances

6.5.1 My Intentions in Live Coding

My general intention is performing extended sets of live algorithmic composition (generative music) with the computer as an active partner. Rather than prepare data and algorithms completely in advance, it was desired that those algorithms or at least their parameters could be adjusted as the music is being performed. I hope to overcome limitations on

musical performance resulting from my body, improvise complex grooves freely, show some of the relationship between sound and code, and provide a novel experience for myself and maybe others.

When improvising, I want to be able to express many different ideas about sound texture and rhythm. I want to be able to improvise without worrying about crashing the software. I want relatively easy access to any system parameter while improvising. I am seeking complex rhythm patterns that repeat in general ways but are constantly varying. The rhythm should be full of syncopation. Rhythm patterns should change at a very high rate in order to prevent them from becoming static. The timbre should be rich and evolving. When multiple tones are present, the harmony thereby composed should not sound incorrect or incoherent. While it is not necessary to strictly follow rules about Western harmony, the chords formed should have some kind of dissonance which makes sense or follow some harmonic principals. The sound samples which are used should be flexible, meaning that they can be paired with relative freedom. There should be few sets that don't work with most of the other samples. The sounds should all be linked in a kind of intuitive or natural sounding way to me.

I hope Conductive is easy to use and facilitates achievement of my musical intentions. I want a music performance system that can also support my musical intentions. An important part of that is being stable when running, free of system crashes. The audience should frequently be able to detect a pulse in the music which they could use for dancing if they choose to do so. I intend for them to be able to try to use the music to achieve their own intentions, whatever those may be. I intend to emit significant volume, particularly bass, through the

sound system. The sound should not be unclear at any frequency range. I intend for the projection to display my activity. I increasingly intend for the code to be readable for the audience. The audience should be able to come to a deeper understanding of how software can be used to create music. At the same time, I intend to display enough of the code on the screen that I am able to easily navigate it while I am editing it in a performance.

6.5.2 Conductive and Other Affectors

A list of affectors can be drawn from the description of my work in section 5. Some of those are likely to be perceived directly, while others are more likely to be perceived indirectly.

Those which affectees perceive directly include things like the samples that I have edited, the rhythm patterns that I generate, the changes in rhythmic density, and the code which I project. They are also aware of the sound system and other aspects of the venue. Unexpected affectors which I have received comments about include my posture at the computer and the movements of my body, people standing over my shoulder and watching me code from behind while others watch from in front of me, and another live coder doing live coding of visuals in conjunction with my live coding of audio.

Those which most affectees perceive indirectly include the audio subsystem of my performance computer, the plugin which allows me to send code from my editor to the interpreter, and many of the abstractions of the Conductive library. Some surprisingly important indirect affectors have included extremely large rooms with strong natural reverberation, the positioning of my set in relation to other people's performances, and fatigue from several consecutive days of performances and travel.

For some of the affectors, it depends on the audience member whether they are perceived directly or indirectly. When I consider myself as an affectee, I perceive some of the abstractions directly at some point and indirectly at all times. For affectees with no programming knowledge whatsoever, it is possible that some abstractions are perceived only from a very indirect experience. It is possible that such an affectee is not even aware that these exist. Some of the code is visible to affectees during a performance, and in this way audience members are able to experience those abstractions. However, most of Conductive is hidden behind higher-level abstractions that I use directly during a performance. Affectees in an audience almost never have a chance in a performance to come into direct experience with the abstractions in Conductive. Even I seldom look at the code in the Conductive library. Rather, I interact with it through its application programming interface, or API. The API defines the parts that are necessary for me to use directly. Even so, I do not use all of the functions pressent in the API, and mostly I only use a fraction of them. The code which most people experience directly is the code which I write in the performance. The following code, part of my Oct. 28, 2014 performance in Ghent, Belgium, is an example of what audience members see in a performance. While a reader now will see it as a block, the audience saw this code come into existence piece by piece as I wrote it. The order is also slightly different, as the pieces defining ensA, testEns, and ensB were written at different places in the file than the remaining code.

```
assignR " 801"

solo [" atmo" ]

playNow e " mid"

playNow e " high"

playNow e " fx"

playNow e " low"

playNow e " sub"


let ensA l n = assignSamples 16 5 [ ([" noisySub" ], " sub" )
                                  , ([" slitheryBassStab" ], " low" )
                                  , ([" gentAtmo-a" ], " mid" )
                                  , ([" bubblyfx01" ], " high" )
                                  , ([" mNoise-02" ], " fx" )
                                  , ([" gentAtmo-a" ], " atmo" )
                                  ]


ensA 16 33 >> solo [" atmo" ]

kitTech 8 9 >> playFull

kitTech 8 9

assignR " 801"

assignR " Lsimp102"


variedGlobalDensity >> ensA 16 33
```

```
let testEns l n = assignSamples l n [ ([" sub2" ], " sub" )
                                    , ([" gliderBass" ], " low" )
                                    , ([" mono-kaos-drone-01" ], " mid" )
                                    , ([" mNoise-02" ], " high" )
                                    , ([" staticfx01" ], " fx" )
                                    , ([" mono-kaos-drone-01" ], " atmo" )
                                    ]


lowGlobalDensity >> testEns 15 9

solo [" atmo" ]


muteHats

playFull >> variedGlobalDensity >> testEns 15 9
```

a selection of code from my Oct. 28, 2014 performace in Ghent, Belgium

This code shows at least 13 different abstractions: assignR, solo, playNow, let, >>, ensA, kitTech, playFull, assignR, variedGlobalDensity, testEns, lowGlobalDensity, and muteHats. Two of the abstractions, “let” and “>>” are from standard Haskell libraries. The rest are functions defined in the main file in which I wrote the code used in the performance. Most were defined at the beginning of the performance. The functions “ensA” and “testEns” were defined during the performance.

The code in this listing is displayed roughly in the order that it was executed. In some cases, a line might have been executed more than once if the results of running the function are different each time. Running a function like “ensA” or “kitTech” changes the character of the overall sound since it changes the samples that are being triggered by the currently running Player processes. The “assignR” function changes the current rhythm pattern, and the “lowGlobalDensity” and “variedGlobalDensity” functions change the graph controlling the global rhythmic density value. The functions “solo”, “playFull”, and “muteHats” change which Player processes are running in ways obvious to the function names. The numbers in the code listing are for parameters such as pattern numbers, pattern lengths, or quantity of audio samples. The strings (denoted by quotations) after the “playNow” function are the names of Player processes. Finally, the “>>” operator allows for the consecutive and immediate execution of functions, such as running “variedGlobalDensity” and then immediately running the “ensA” function.

6.5.3 Experiencing a Particular Live Coding Performance

I recorded a screencast of a 15:15 performance on Dec. 9, 2014. It is in a 170 bpm experimental drum and bass style and could be classified as algorave.

The intention behind the performance was to make a video in which the code could be clearly seen. I also wanted to upload a video to my Youtube channel with audio of a better quality than what was there at the time. Finally, I wanted a performance that could be analyzed for this document. Beyond these intentions, the intentions described in section 6.5.1 were also targets.

Earlier tests of my typical font at a small size yielded videos in which the code was not legible at small sizes. As a result, this video used a larger size of VL Gothic in place of the 12 pixel Terminus that I typically code in. I also gathered some arrangements of samples in a separate file of code so that sample sets could be changed more quickly. Finally, I increased the tempo up to 170 from the 140 and 160 that I have been performing at recently. The tempo was increased from the influence of music that I had been listening to and to give the music more energy. I felt that I could more safely make many changes in the music if it was moving at a faster tempo.

The screen capture software is briefly shown before the video cuts to the view of the interpreter and text editor. There are three function calls on the screen: a line to pause all Players, a set of sample assignments, and a function to load a drum kit.

I begin adding to the line related to the drum kit at four seconds. Some additional functions are one to set the global density value envelope and a function to start only the Player

called “atmo”. This line is executed and appears at the bottom of the interpreter window. The music starts with a drone at 0:17, and the interpreter responds with output.

A new line is written to start another player and select a rhythm pattern. That is executed at 0:44 and a percussive bass starts. After it is executed, two additional Players, triggering hi-hat and sub-bass samples, are started at 1:19. The samples being played from the kit are refreshed and the global density envelope is changed to “varied” from “low” at 1:41. At 2:15, all of the Players except for the drone are stopped. The drone proceeds by itself until 2:26, when the full ensemble of 12 Players begins to run together with a new set of samples.

The function “assignR” is used to load rhythm patterns, like 801 and Lsimp101. At 2:46, some improvisation takes place with rapid switching between these two rhythm patterns. At 3:20, more of such improvisation, along with switching of ensemble and drum kit sample sets. As a rhythm pattern is changed, the interpret flashes as it fills with the contents of the new rhythm pattern.

At 3:33, the number of active Players is reduced by using the “stripDown” function. At the same time, the global density is increased. This has the effect of making the sound feel closer and more intense but not quite a wall of sound. Until 4:00, some function calls are written and finally a function calling a new set of samples is loaded into the interpreter. The sample set is changed at 5:04, and the samples are switched to those from a new ensemble at 5:25. The dark kit is refreshed several times until the Kperc kit is started at 5:48 along with a new drone noise which plays in the background.

New rhythm patterns, Lsimp102 and 802 are used for rhythmic improvisation around 6:00. Other rhythm patterns used later in this performance include LBSimp 102, 401, 402, 403.

All of the rhythm patterns were generated while setting up for the performance video.

Using a higher-level abstraction built on top of the abstractions for density and L-system-based pattern generation, a 99 patterns are generated based on the L-system “a:ab, b:ac, c:aac” with input of “c” and generated up to the sixth generation. The preparation of this code begins at 6:06, and this is executed at 7:13 to generate the 99 related patterns and their density variations. The entire contents of the rhythm pattern store is printed in the interpreter. Several patterns from this set are auditioned in the performance, including L1101, L1102, L1103, and L1104.

Pattern 401 was listened to briefly after being loaded at 8:48, before I move to pattern 402. Pattern 401 had too little syncopation, so it was quickly changed. The density is increased at 9:00. A new sample set assignment function is defined at 9:03, and this is executed at 9:10. The sound is stripped down at 9:24. The hats are added back at 9:34. Next, ens4 is run. Altogether, there have been five sets of samples: ens0, ens1, ens2, ens3, and ens4. There have been four drum kits: dark, Kperc, WaveTom, and BKp.

All of the samples are switched with the rhythm pattern at 9:56. At 10:07, the sound is made very intense with all Players playing at a high density. More improvisation takes place around 10:29, largely using pattern 402. The density is fairly high at 11:00.

Ens4 is prepared at 11:30, and there is a big change in the sound at 11:55. Samples are swapped at 12:12. The samples are changed again at 12:35, and again repeatedly. Each time, the number of sample used in the sample loop is result, adding to the intensity of the building peak. Improvisation takes place with alternation between L1103, L1104, and 403. The global density is increased again at 13:30.

The busy drums come to a stop at 13:52 with the execution of lowGlobalDensity and stripDown. The music ends with something similar to a reprise of the drone and sub-bass stabs at the beginning of the performance, with the stabs finally dropping out and leaving the drone alone. The sound is stopped by executing pauseAll, with the sound dying out by 15:14.

Affectors in this performance include most of the affectors described in previous sections. The video of the screencast takes the place of projection, and naturally the venue and audience vary according to the environment that the viewer watches the video in. There is the characteristic visual effect of the left side frequently full of numbers while the right side has more negative space. The exact nature of the audio depends on the listener. While performing and evaluating this video, I was wearing headphones. Additional affectors include the improvisation that takes place when alternating between rhythm patterns and the pace of changing sample sets. The abstractions which function directly here for myself as affectee are, among others, higher-level abstractions built upon the rhythm pattern and density functions described earlier. The rhythm abstractions in particular are the density functions, the “assignR” pattern for switching rhythm patterns, and the function which was used to generate additional L-system-based patterns.

Several of the original intentions were satisfied: a video could be made in which the code could be clearly seen. I was able to upload this video to my Youtube channel with reasonably high-quality audio, and I was able to analyze the video for this document. Improvisation was possible, and a variety of rhythm patterns and timbre sets were presented.

Some of the affects experienced include frustration, surprise, satisfaction, and expansion. The frustration came from the fact that the sample arrangements had to be coded before the

performance was carried out. The surprise was pleasant surprise that the video felt more like an effective improvisation than I expected and that the 170 bpm tempo worked well despite being my first time to improvise with this system at that tempo. The points providing satisfaction are essentially the same as those providing surprise. The sense of expansion comes from completing this video. Since 2011, my videos have all been from a camera and taken during a live public performance. This is the first screencast that I had done since 2011, and it was a satisfying feeling to do a new one.

It is hoped that some better and less time-consuming method can be found for composing these sample sets. In addition to revising the sample handling, it will be nice to have some chances to do DSP programming in a set. Despite the relative freedom experienced in improvising with rhythms, I still felt it would be preferable to have some additional algorithms for the generation of rhythm patterns and density patterns. The font weight and shape is different from what I usually use. It was effective for achieving decent character shapes at that size, but I would like to try other fonts to get an ideal choice for performances. Having done this screencast, I am now encouraged to do more such videos.

6.6 General Evaluations from My Live Coding Experience

The Player data structure has been very useful in testing over the past two years. The flexible usage of action functions and IOI functions appears to make editing safer and more convenient, from the author's experience. This arrangement also made group operations on multiple Players possible, opening the possibility for abstractions that deal with sets of Players. It is a particularly appropriate abstraction if the user employs a representation of

music similar to that of a traditional musical ensemble, like a four-piece band. However, it is not limited to such models, preserving a high degree of generality.

Changing from MVars to TVars with STM is thought to have solved some mysterious runtime misbehavior.

The generality of TimespanMaps also appears to be high. It proves through usage to be a flexible way to represent time, and was seen as a useful tool even in places it was not initially intended for use. That is the result of the widespread occurrence of values that change with time.

By using TimespanMaps with the sampler, it was possible to reduce the number of players for 70+ samples from 70 Players to between four and eight Players triggering hundreds of samples. This arrangement was found to be much more manageable and sonically-attractive than the previous one. It was very difficult previously to look at the list of Players and see which ones were playing and which ones were not. It also made changing the arrangement very hard, as Player-related functions often required long lists of Players. The current arrangement still uses lists of Players at times, but the lists are much shorter.

In the system described above, parameters for synthesis are initiation-rate values. That means that the timbre of a particular event does not change over time other than what is contained in the original sample.

The lack of continuous timbral modification through effects is a sore spot. A moving timbre can make the sound much more lively, but this is possible to a limited degree in the system above because of the design of the current sampler. Varying the sample of a particular Player does change the timbre, but sometimes a change which unfolds in a discernible direc-

tion over time can be a more effective compositional device. This has only been achieved in the current version for the sample pitch and amplitude.

Varying rhythm became much easier. It became easier to control performances. The results became less random and more musical. Performances are more interesting than they were previously.

This particular implementation of an abstraction for density, on the other hand, is linked to a particular algorithmic composition technique. As such, while the underlying notion is a general one, the implementation is too specific to one compositional style. Specifically, the method for generating the variations of the input pattern was fixed. A more abstract version in which the method of determining the contents of the variations can be supplied from outside is needed.

Patterns from the pattern generator are processed into density maps, with interpolated density values held in TimespanMaps. A Player then uses an IOI function designed to retrieve an IOI using that stack of abstractions. This enables relatively fast generation of large amounts of rhythmic material and variations.

Using ratios for increasing density works fairly well as long as the ratios are very simple, like 0.5. Other ratios generate patterns that are likely less familiar to listeners, and thus might not be appropriate if the composer has the intention of producing music that neatly fits within most existing genres. However, this was just an example, and more sophisticated methods can now be more easily tested.

The use of L-systems is also interesting, but it will take additional practice to become acquainted with L-systems and how, in the middle of a performance, to write rules and axioms

for interesting output. As with the density function above, these L-system functions are also simple examples that can be refined or replaced for future work. They simply demonstrate that generalization was possible.

The complexity can also be hard to keep in one's head and manage. It is challenging to keep a mental model of the parts described above during a performance, even though what has been described is mainly concerned with the timing of events and not timbre. This suggests that adding the complexities of generating different timbres through synthesis during performance will be burdensome. Part of this burden can be overcome through more practice with the system, but it seems that there is still a higher-level of abstraction to be achieved for optimal usage.

At the same time, audiences seem to have limited understanding of how the abstractions work, and the font size at which they have been projected has further prevented their understanding.

Current methods for organizing the text data or source code used during a performance are poor. As a result, the text in the text editor quickly becomes messy in the course of a performance. That makes it harder to stay in control of the performance or to run previously-defined functions at the most ideal times. Maximum effectiveness of use of the editor environment probably has not yet been achieved. Editor usage skills or tools to aid in this are probably needed.

However, it is thought that even these implemented abstractions improved the usability of the associated live coding system, enabling performances that would have otherwise been much more difficult or even unachievable without them. The approach above makes it

possible to perform for extended periods of time, mostly limited by the amount of samples that have been prepared in advance.

7 Conclusion

7.1 Summary

This document has developed the following areas:

The introduction of this paper provided an introduction to live coding. It explained the means for doing so, and gave some justifications for live coding. That was followed by a problem statement on how to evaluate abstractions for rhythm in live coding using aesthetics.

The next section of the paper gave the necessary background information for pursuing that problem statement. It began with a brief history of live coding. It then necessarily moved to a discussion of abstractions, rhythm, and abstractions for rhythm in live coding.

The following section described my work, which was a practice-based means to answering the problem statement. The first work was practical musical theory about rhythm. From that theory, software implementing those ideas was described. Finally performances using that software were presented. The music theory has been validated by its ability to be implemented as software and affective response to its output. The software has been validated by its potential to be used in performances and affective response to using it. The performances have been validated by their acceptance to conferences, festivals, and other performance opportunities, as well as their affective characteristics.

The last section before this conclusion was a discussion of a pragmatic aesthetic theory and its application to abstractions for rhythm in live coding. It analyzed the practice-based work produced in this research from the perspective of that theory. It included both a descriptive pragmatic view of how art experiences are constituted and a normative theory on how to

evaluate such experiences. Both theories are supported by their instrumental value.

7.2 Results

The contribution of the theoretical work and software development has been acknowledged by the acceptance of three papers to peer-reviewed conferences. The use of the implemented software in performing a series of concerts both domestically and abroad has been an achievement. Altogether, 20 performances have taken place in 11 countries. The theories presented in the discussion section have been accepted as papers to three peer-reviewed symposia, seven peer-reviewed conferences, and one peer-reviewed journal.

This research contributes the following to the field:

1. a definition of abstraction and an argument for its centrality to live coding
2. some music theory and techniques for the generation of rhythm patterns
3. some tools for the generation of rhythm patterns in the Haskell programming language based on the music theory and techniques mentioned above
4. a series of performances, both domestic and abroad
5. a restatement of Dewey's aesthetic theory, leading to an affect-based pragmatic heuristic for the evaluation of art experiences
6. an analysis of the many facets of live coding
7. an argument that abstractions can be evaluated aesthetically
8. applications of the pragmatic heuristic to live coding, in particular abstractions for rhythm in live coding

7.2.1 Music Theory

In the area of music theory, the following contributions were made. A stochastic method for generating rhythm patterns was described. An additional method for generating rhythm patterns combining stochastic techniques with L-systems was also described. An algorithm for variations of a rhythm based on density was explained. The pattern generation methods were paired with the density-based variation technique to provide algorithms for generating complex rhythmic sequences.

7.2.2 Music Software

Software has been developed which makes a significant contribution to the field. Conductive is one of the earliest libraries for live coding in a general purpose programming language. It is a very flexible framework providing usable abstractions representing the music theory contributions described above.

7.2.3 Performance

Performances have presented a fast-paced dynamic sound that reflects the agility made possible by the Conductive library. They feature distinctive rhythms and sound design, and as a result these performances have been accepted by eight peer-reviewed conferences on the criteria of technical execution, novelty, and aesthetics.

7.3 Final Conclusions from this Research

Through this research, it has been concluded that it is possible to make well-reasoned evaluations of the abstractions for rhythm which are used in live coding. Those evaluations can be made through a pragmatic affect-based aesthetic heuristic. Such evaluations can be used to satisfactorily improve abstractions.

7.4 Future Directions for This Research

This research illuminates many new paths to pursue. This is only a partial list of future directions for the research presented in this document. Many other possibilities are imaginable.

The current algorithms can be refined. It may be possible to add additional parameters to give them more flexibility.

More types of algorithms for rhythm pattern generation and density should be explored, both algorithms from other fields which can be repurposed for music and novel algorithms specifically designed for musical purposes.

The algorithms described by this research were for rhythm pattern generation, typically between one and 16 musical bars. Additional higher-level abstractions for musical form should be developed. These abstractions would deal with musical structures longer than 16 bars. The advice of Berg[332] and the theories of Xenakis[333] should be examined for applications to this system.

[332] Berg, “Abstracting the Future.”

[333] Xenakis, **Formalized Music**.

These techniques should be applied to algorithmically generated visual art.

In the area of aesthetics, the heuristic could be further refined based on the writings of Russell and Panksepp. This heuristic should be used for the development of abstractions for other purposes than rhythm. In particular, it might provide insights into the development of abstractions for timbre and harmony. It is likely also possible to use this heuristic for the development of visualizations. Research could be done on the application of this heuristic to other media, such as visual art, media art, other types of performance art, and so on.

Finally, it is believed that an effective live coding system will more deeply employ the computer as more of a partner in a performance. This means taking advantage of artificial intelligence techniques. Initially, the Player abstraction in Conductive could be used for conducting other Players if the necessary action functions were designed. This area should be explored to give the coder more freedom between code executions. It is hoped that both this library and the aesthetic theory described can be employed to test much more advanced AI techniques. The library may be a useful infrastructure in which to employ agents or other AI techniques. The aesthetic theory, if pushed further toward an algorithm, may form the basis of a project in computational aesthetics.

8 Appendices

8.1 A List of Performances

The following solo performances are listed in chronological order. All are single performances, with the exception of the July 12, 2014 concert, which included two performances.

1. 2013, May 11: Linux Audio Conference 2013 in Graz, Austria (performance)
2. 2013, June 15: Algorave at Musical Metacreation Weekend (MUME) 2013 in Sydney, Australia (performance)
3. 2013, June 19: 9th ACM Conference on Creativity & Cognition 2013 in Sydney, Australia (performance)
4. 2013, November 14: SI13 in Singapore (performance)
5. 2013, December 11: Generative Art Conference in Milan, Italy (performance)
6. 2014, January 05: “algorave ++ noise” at Nanahari in Tokyo, Japan (performance)
7. 2014, April 05: RANDOM() at Soup in Ochiai, Tokyo, Japan (performance)
8. 2014, May 03: Linux Audio Conference 2014 in Karlsruhe, Germany (performance)
9. 2014, June 27: Algorave at xCoAx (conference) in Porto, Portugal (performance)
10. 2014, July 03: Algorave at New Interfaces for Musical Expression (conference) in London, UK (performance)
11. 2014, July 04: Algorave in Brighton, UK (performance)
12. 2014, July 06: Algorave in Sheffield, UK (performance)
13. 2014, July 07: Algorave in Leeds, UK (performance)
14. 2014, July 08: Algorave in Manchester, UK (performance)

15. 2014, July 09: Algorave in York, UK (performance)
16. 2014, July 12: live coding at Landbouwbelang in Maastricht, Netherlands (two performances)
17. 2014, October 18: live coding at La Escucha Errante 2014, Oct. 18, 2014, in Bilbao, Spain (performance)
18. 2014, October 28: Lambda Sonic Algorave, in Gent, Belgium (performance)
19. 2014, October 29: live coding at Studio 539 Music Night in Amsterdam, Netherlands (performance)

8.2 Details of the Conductive System

Overview

This library exists to wrap concurrent process manipulation in a way that makes controlling their timing more intuitive for musicians. At the same time, the library aims at being as concise as possible to lessen the burden on the user. The library, called Conductive, was programmed in Haskell. The Haskell programming language was seen as a good candidate because of several factors: expressivity, speed, static type system, large number of libraries, and ability to be either interpreted or compiled. It lacked a library suitable for this author for realtime manipulation of musical processes. McClean is also developing Tidal,[334] a Haskell library with a similar aim.

It contains abstractions for musical time, musical events, and event loops. This gives the Haskell interpreter the ability to function as a code-based real-time sequencer for any output targets which can be connected to the system. Conductive does not aim to be an audio language, but a controller for audio output targets. The user is free to choose any OSC-aware output target, and this library is proposed as a way to coordinate those outputs. Another way to think of it is as a shell or scripting environment for real-time music.

Conceptually, it has similarities with the concepts in SuperCollider of Routines, Tasks, and Patterns. Some key similarities and differences are noted below, along with details on each of these components.

Summary of Conductive Concepts

[334] McLean and Wiggins, “Tidal–Pattern Language for Live Coding of Music.”

Some basic concepts for using Conductive include the notion of Players, action functions, interonset interval (IOI) functions, and TempoClocks. These concepts were originally explained in a paper from 2011.[335]

Players are representations of concurrent processes that perform actions separated by periods of time called interonset intervals (IOIs). A Player runs its specified actions and then waits for an IOI determined by its specified IOI function. This loop is instantiated by employing the “play” function with a Player as an argument.

Actions functions define what is done by a Player. These actions could include triggering a synthesis event or modifying the general system state. The only limitation is their type signature, since Haskell is a statically-typed language. This means that the types of arguments to an action function are fixed, and they must return the unit type in the IO monad, or “IO ()”. Currently, a sampler action is used predominantly.

IOI functions define how long to wait between actions. Any methods available to the programmer could be used to generate those times, from simply returning a value, such as one second, every time, to table lookup of values, to the calculation of values based on complex mathematical formulae.

The user defines TempoClocks which have a tempo and time signature.

These concepts are described in greater detail below.

Utilized Tools from Other Developers

In order to use this system, there are some prerequisites.

[335] Bell, “An Interface for Realtime Music Using Interpreted Haskell.”

The first of those is a Haskell programming environment. The Glasgow Haskell Compiler, which contains an interpreter (GHCi) that allows the interactive evaluation of source code,[336] is used by the performer to call functions from the Conductive library. The process of writing source code and sending it to GHCi is made more usable with vim (a text editor),[337] tmux (a terminal multiplexer)[338], and a vim plugin called tslime that allows text to be sent from the editor to the interpreter through tmux.[339]

To do live coding, a library called Conductive,[340] written by the author in the Haskell programming language,[341] is employed with hsc3 (the Haskell bindings to the SuperCollider synthesis engine)[342] on top of a standard Linux audio system (ALSA and JACK) with Patchage[343] for routing.

As Conductive does not directly handle sound synthesis, a method for synthesizing sound is necessary. This paper describes the use of the scsynth component of the SuperCollider package.[344] At present, synthesis events are programmed in Haskell and employ Rohan Drape's hsc3 Haskell library for communicating with scsynth.[345] A sampler (described below)

[336] Jones et al., "The Glasgow Haskell Compiler."

[337] Moolenaar, **The Vim Editor**.

[338] Marriott and others, "Tmux."

[339] Coutinho, "Tslime."

[340] Bell, "An Interface for Realtime Music Using Interpreted Haskell"; Bell, **Conductive-Base**.

[341] Jones, **Haskell 98 Language and Libraries**.

[342] Drape, **Haskell Supercollider, a Tutorial**.

[343] Robillard, **Patchage**.

[344] McCartney, "SuperCollider."

[345] Drape, **Haskell Supercollider, a Tutorial**.

uses samples that have been generally recorded and edited using Ardour,[346] and they have largely originated from hardware synths. All of the samples are individual sounds, from single-shot percussion sounds to bass samples. Most are wav files under 300 K.

Finally, in order to achieve a solid sound closer to that of commercial releases or broadcasts, the output of scsynth is processed through Calf plugins hosted by the Calf stand-alone host.[347] An EQ is followed by a multiband compressor and then a limiter, whose output is directed to the soundcard. Output is also directed to JAAA for monitoring.[348] Patchage is used for ease of routing.[349] Recording of performances is done with either Ardour (in the case of audio) or gtk-recordmydesktop (in the case of video).[350]

TempoClock

The tempo is part of a TempoClock, a concept from SuperCollider which is reimplemented here in Haskell. A TempoClock is like a metronome keeping the current tempo but also containing information about time signature and when tempo or time signature has been changed.

A TempoClock is a record of the time the clock was started, a list of TempoChanges, and a list of TimeSignature changes. This allows a user to independently manipulate both tempo and time signature and to use these for composing and performance in addition to regular

[346] Davis, **Ardour**.

[347] Foltman et al., **Home @ Calf Studio Gear - Audio Plugins**.

[348] Adriaensen, **Kokkini Zita - Linux Audio**.

[349] Robillard, **Patchage**.

[350] Varouhakis and Nordholts, **recordMyDesktop Version 0.3. 7.3**.

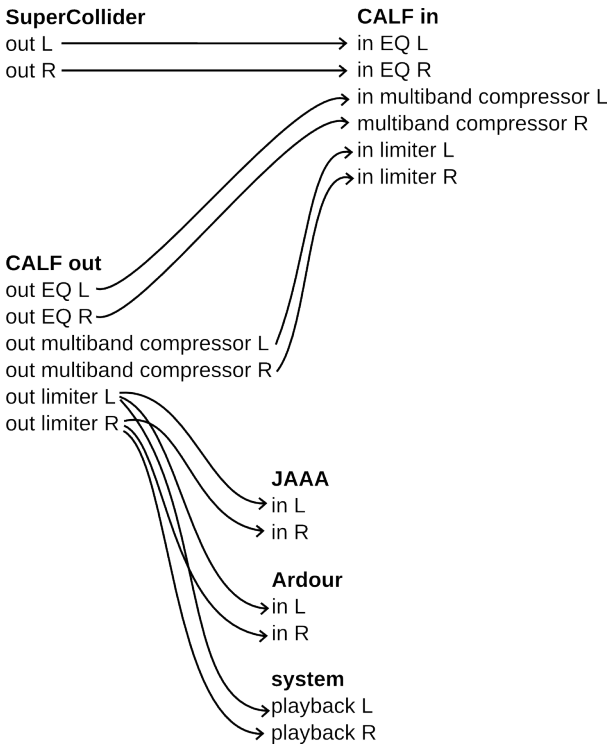

Figure 2: signal flow

POSIX time.

TempoClocks are stored in the MusicalEnvironment.

MusicalEnvironment

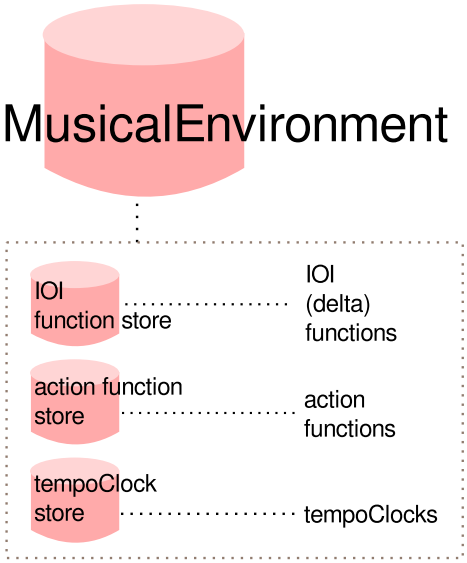

Figure 3: MusicalEnvironment: a data structure for storage

The MusicalEnvironment is a place to store data which is used by any of the user-initiated event loops. This data consists of:

- the name of the environment
- a store of Players
- a store of TempoClocks
- a store of IOI functions
- a store of action functions
- a store of interrupt functions

Players

The programming pattern of defining processes and having them repeat at various intervals seemed a fundamental one and a good target for abstraction. An abstraction called a Player was developed to manage these concurrent processes.[351] An analogy for a Player could be a solo performer or a person in an ensemble, or it can be thought of like a media player.

A Player initiates a process, waits for a period of time, initiates another process, waits for another period of time, and repeats this as long as it is running. The processes that a Player initiates are described in action functions. At present, the action functions in this author's performances are triggering sample-playing synthdefs in scsynth (SuperCollider synthesis engine).

It seemed desirable to create a separation of concerns in which the events and their timing could be dealt with independently, one changed without affecting the other and in which one, the other, or both could be reused by other concurrent processes. Another benefit of this separation is increased role-expressiveness, where the function for time is clearly labeled as an IOI function, as is the function generating events (as an action function).

Players was designed as a way to sequence IO actions. Players contain references to particular data which is stored in the MusicalEnvironment. The collection of data referenced by the Player results in a series of actions being produced once the Player is played. This data consists of:

- the name of the Player

[351] Bell, "An Interface for Realtime Music Using Interpreted Haskell."

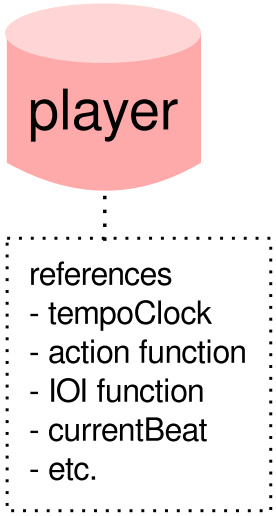

Figure 4: Player: a data structure filled with references

- its status (stopped, playing, pausing, paused, stopping, resetting)
- a counter of how many times it has run an action
- which clock it is following
- which IOI function it is using
- which action function it is using
- which interrupt function it is using
- which beat its next action occurs on
- which beat it started on
- the POSIX time at which it was last paused

Players bear some resemblance to Tasks or Patterns in SuperCollider; they can be played, paused, and stopped to produce music. However, while Patterns in sclang can produce streams of any kind of data, Players in Conductive are designed to produce streams of

side effects. While the data in a Pbind in SuperCollider is generally private [?], all the data contained by a Player is visible.

Players are stored in the Player store, a mutable key-value store where the keys are Player name strings and the values are the Players themselves. This in turn is part of the MusicalEnvironment. How patterns are stored in SuperCollider is up to the individual user. This library provides a readymade structure for that purpose.

To briefly describe some aspects of this abstraction, using a data structure called Player as an argument, a recursive loop instantiated by the “play” function forks events (IO actions, such as triggering synths and changing program state) and waits. The exact action and wait time are according to referenced action functions and functions that return the timing of the next event. This effectively decides the interonset interval (IOI) until the next loop, on which the action function and IOI function are freshly retrieved. These functions are stored outside the Player data structure, which only contains references to the stored functions.

Players bear some resemblance to Tasks or Patterns in SuperCollider; they can be played, paused, and stopped to produce music. However, while Patterns in sclang can produce streams of any kind of data, Players in Conductive are designed to produce streams of side effects. While the data in a Pbind in SuperCollider is generally private,[352] all the data contained by a Player is visible. Players are considered to be higher-level than Tasks or Patterns.

Action Functions

[352] Harkins, **A Practical Guide to Patterns**.

An action function is a function that describes an event. An action function outputs a value of the IO unit type. This basically means some kind of side effect is produced without returning a value like a double or a string. In practical terms, this could be a synthesis event, a parameter change, or the action of playing, pausing, or stopping other Players or itself. It is thought that the user would use functions which send OSC messages to connected OSC-aware applications. The action named in the Player can only take two parameters: the Player triggering the action and the MusicalEnvironment it should read from. Beyond that, the design of the action is left to the user. A user might prefer to have many Players with simple actions, a few Players with complex actions, or some other combination.

Interrupt Functions

An interrupt function is a function which is run once every time the play loop runs. It is useful for debugging purposes, and may be used to trigger other actions, such as stopping the player on a condition.

Interonset Intervals (IOIs) and IOI Functions

A fundamental concept is that of the time interval between the start times of two events, or interonset interval (IOI).[353] SuperCollider refers to this as “delta” with regard to Patterns or “wait” for Routines. The IOI is defined in beats, and the actual time between events is calculated using the IOI value and the TempoClock referenced by the Player it is associated with. IOI functions should also be designed to read the data from a Player and a MusicalEnvironment. They can be designed in any way the user desires, including always returning a particular value, stepping through a list of values stored in a list somewhere, randomly

[353] Parncutt, “A Perceptual Model of Pulse Salience and Metrical Accent in Musical Rhythms.”

choosing a value, or anything else the composer can imagine.

The time that a player waits between initiating events is called an interonset interval (IOI). It is possible than an event has not finished running when the next event starts. An IOI is expressed in terms of beats, which in turn are based on a TempoClock. This IOI is determined by an IOI function which the Player refers to. It calculates the next beat that it should run its action function on.

A simple technique employed in an IOI function is looping through a list, successively returning each value as the current IOI, but the design is up to the user. Writing such lists by hand during a performance took too much time. As a result, abstractions were designed to make the generation of such lists easier.

A stochastic generator of IOI lists was developed which first makes a set of potential IOIs based on a core value. It then generates a set of subphrases based on those potential IOIs. It outputs a final phrase by choosing from those subphrases up to a user-specified length of time. This uses the concepts of repetition and variation to create arguably aesthetically-pleasing patterns.

Play

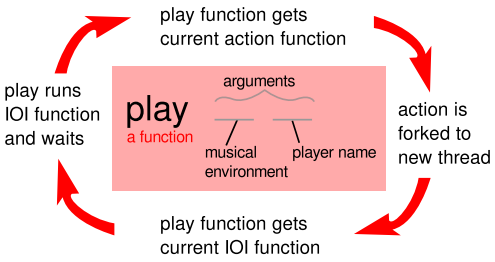

Figure 5: The play event loop

The play function starts a thread which forks other processes according to a schedule determined by the IOI function referenced in the Player. It takes a MusicalEnvironment, a Player store, and a Player name as arguments. First, the play function checks which action is referenced in the Player. It retrieves that function from the MusicalEnvironment and forks it to a thread. It then checks which IOI function is referenced in the Player. It runs that function and receives a numeric value specifying how long to wait in terms of beats. It then corrects that amount for jitter and sleeps for the corrected length of time. When the thread wakes up, the loop – checking the action and so on – repeats.

It produces roughly similar results to calling play on a Pattern in SuperCollider in that it begins a process; however it is structured differently.

The problem of dealing with the delays in scheduled events is significant. Because various processes, including garbage collection, can conceivably interfere with correct timing, correction of jitter is included in the play event loop. This library does not introduce a novel method for managing such delay, but rather adopts a design from McLean.[354] An event intended to occur at time x actually occurs at time x + y, where y is the amount of time by which the event is late. The next event is scheduled to occur at time x + z, where z is the IOI, so to account for the jitter, the wait time is set for x + (z-y). In practice, this delay is generally permissible for control data, while it would not be appropriate for audio data.

The number of simultaneous play event loops is limited only by the memory and CPU of the host machine. Since at every loop the data used is refreshed, they can be manipulated in real time by changing the data stored in the Player or MusicalEnvironment. Which action

[354] McLean, “Hacking Perl in Nightclubs.”

function or IOI function is referenced in a Player can be changed. The action functions or IOI functions themselves can be modified. Any of the other data in the Players or MusicalEnvironment can be changed. By changing this data, the resulting musical output can be changed. It is in this manner that a livecoded musical performance is realized.

Such manipulation results in many threads and the potential exists for one thread to be writing data which is accessed by another. One problem of real-time multi-threaded systems is guaranteeing the thread safety of data. Haskell provides safe concurrency in the standard libraries of the Glasgow Haskell Compiler (GHC).

An Example of How Players Work

Here is an example of how Players work, shown in figure 6.

Consider a timpani Player called “A” who has only two jobs. The first job is to hit the timpani. The second job is to wait for a given amount of time, like that written on a score. He hits the timpani, then he waits, then he hits the timpani again and waits, in a loop until he is asked to stop. Now imagine that this Player is joined by another: Player “B”. The second Player has only two jobs. The first is to adjust the tuning of the timpani; the second job is the same as that of the first Player. He tunes the timpani and waits, and then tunes it again and waits, repeating like the first Player.

The first timpani Player is a Player stored under the key “A” in the Player store. Its action function is “hit the timpani”, which may correspond to triggering a synthdef on scserver called “timpani”, which results in a timpani sound being played. The second Player is called “B”, and its action function, “tune timpani”, is to change the frequency parameter used by the “hit the timpani” function. Each of them has its own IOI function.

Let's expand the situation to include two more Players, Players "C" and "D", who correspond to Players "A" and "B" but are involved with another timpani. The resulting sound is two timpanis being played at the same time. In this case, the "hit the timpani" action is designed to use the name of the Player to determine which frequency should be used. In the same way, the "tune timpani" function uses the Player name to determine which frequency it is tuning and which frequency to tune to.

Now, interestingly, we'll add a fifth Player, who is starting and stopping the Players above. Its action function cycles through a list of actions. Its first action is to start Player "A". Its second action is to start Player "B". Its third action could be to start Players "C" and "D" simultaneuously. Its fourth action could be to pause Players "A", "B", and "D". The design of any action is up to the intentions of the musician.

TimespanMaps

Another commonly observed pattern was setting the timing of particular values. It is often desired that values change over time but at different rates. TimespanMaps are structures for handling such cases. Rather than specify the exact timing of a value, it specifies the range of time in which that value can occur. They are maps or dictionaries with intervals as keys to any kind of value. Another parameter of the structure is a specified length at which it loops. When a time is passed to the dictionary, the interval that time falls in is determined to be the key to use, and the corresponding value for that interval is returned. When the time value passed to the TimespanMap exceeds those for which it is defined, it loops to return an appropriate value.

Convenience functions for TimespanMaps with random key values and interpolated Times-

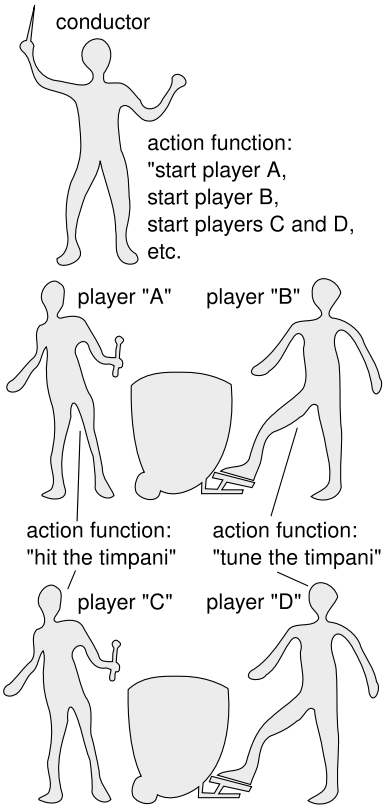

Figure 6: An example of Players at work

panMaps (in which fixed-length steps are linearly interpolated from a set of points in time) were developed to increase the speed with which TimespanMaps could be created.

Another commonly observed pattern was that of setting the timing of particular values. It is often desired that values change over time but at different arbitrary rates. An abstraction was sought to handle such cases.

A TimespanMap is in a way a more-abstract event list, except that it can be used for any type of value rather than just events. Rather than specify the exact timing of a value, it specifies the range of time in which that value can occur. They are maps or dictionaries with intervals as keys to any kind of value. Another parameter of the structure is a specified length at which it loops. When a time is passed to the dictionary, the interval that time falls in is determined to be the key to use, and the corresponding value for that interval is returned. When the time value passed to the TimespanMap exceeds those for which it is defined, it loops to return an appropriate value. The rate of change in a TimespanMap is arbitrary and can change within the map. That is, each interval can be of an arbitrary length, and the number of items in a TimespanMap is limited only by memory or performance constraints.

When TimespanMaps are used in conjunction with the density maps described above, the values are either density values (for determining which IOI pattern to use) or an IOI value to be selected from a particular IOI pattern. They can be used for any parameter, not just those above, such as determining the amplitude or pitch of an event.

TimespanMaps are maps or dictionaries with intervals as keys to any kind of value and a specified total length for the whole TimespanMap. In the case of IOI pattern tables described above, the values are either density values (for determining which IOI pattern to use) or an

IOI value to be selected from a particular IOI pattern.

A time in beats is passed to the dictionary. The interval that time falls in is determined to be the key, and the corresponding value for that interval is returned.

TimespanMaps can be used for any parameter, not just the ones described above. For example, in this system it also used for determining the amplitude and pitch of a particular triggering of a sample, as well as which sample to trigger.

The rate of change in a TimespanMap is up to the user and can change within the map. The number of items in a TimespanMap is limited only by memory or performance constraints. The range of time covered by a particular key can be as large or small as the user determines to be appropriate and is limited only by the Double data type in the Haskell language (double-precision floating point number).

A sample TimespanMap with a time length of four might look like this:

- length: 4
- 0: “a”
- 2: “b”
- 2.5: “c”

When passed a time of 0, the TimespanMap returns “a”. With 0.5, “a” is also returned. With a time of 2.25, “b” would be returned, and with a time of 3.5, “c”. If passed a time of 4, the list loops and “a” is returned.

Figure 2 shows the relationship between IOI patterns and density tables. It includes one TimespanMap mapping intervals to density values. Missing from the illustration (in order to

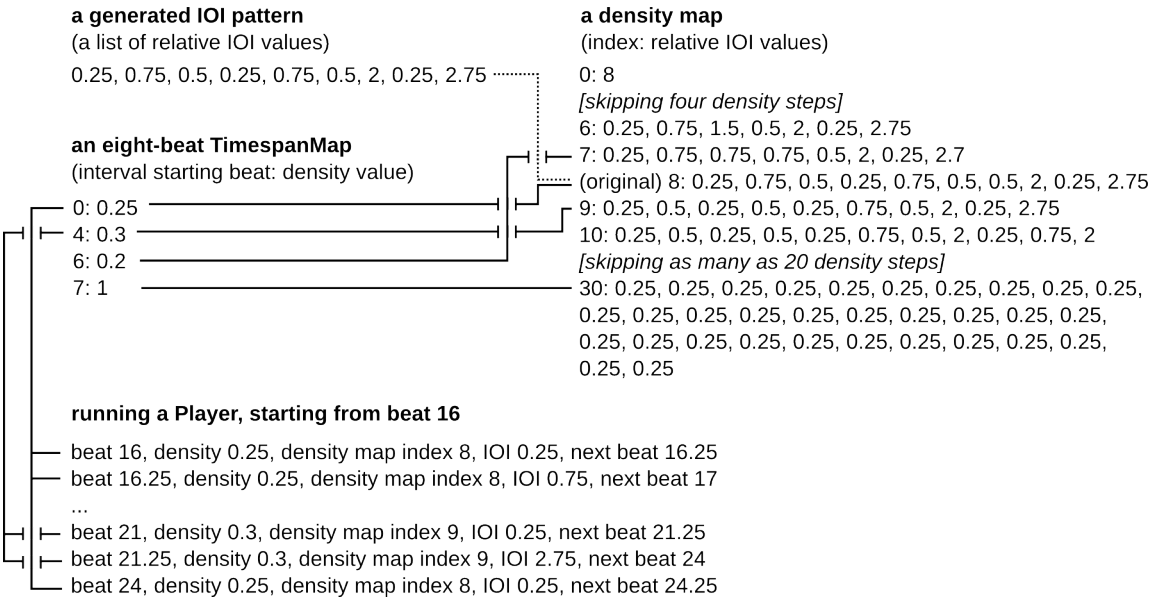

Figure 7: an example showing the relationship between IOI patterns, density tables, and TimespanMaps, based on the example from section 4.1

keep it less cluttered) is the fact that the IOI patterns themselves are TimespanMaps from intervals to IOI values. The figure shows the calculation of the next IOI value of a running Player from beat 16 to beat 24. The example does not show the complete contents of the density map in order to save space, but actually generating a density map provides the full range of IOI patterns.

Convenience functions for TimespanMaps with random key values and interpolated TimespanMaps (in which fixed-length steps are linearly interpolated from a set of points in time) are provided in the library.

In addition to using TimespanMaps with IOI values and in density maps, they have been used for samples. Previously, one Player was assigned to each sample. In order to use 70 samples, it was necessary to instantiate 70 players. Managing 70 Players was challenging,

so the sampler was rewritten to employ a TimespanMap. Subsets of sample sets are chosen at random and one is assigned for each interval in the map. When the sampler is triggered, the sample is chosen according to the current beat. By doing so, the number of Players needed was reduced by roughly a factor of 10.

8.2.1 Code Examples of Conductive Usage

A rudimentary sample of usage and corresponding code is given below.

First, the relevant Haskell modules must be imported, which is accomplished by loading a Haskell document containing the necessary import statements.

```
:load Conductive.hs
```

This example uses SuperCollider, so a convenience command which sets up a group on scserver is called.

```
defaultSCGroup
```

A default MusicalEnvironment is instantiated. It is assigned to the variable “e”.

```
e <- defaultMusicalEnvironment
```

An scserver-based sampler is instantiated using this command, which also creates the necessary Players and action functions in the MusicalEnvironment. The function takes a path and the MusicalEnvironment as arguments.

```
s <- initializeSampler ″ ../sounds/*″  e
```

All of the Players in a specified MusicalEnvironment can be started with the playAll function. The argument, like above, is the MusicalEnvironment.

```
playAll e
```

The status of all the players in a specified MusicalEnvironment can be viewed with the displayPlayers command.

```
displayPlayers e
```

A list of players can be paused using the pauseN function. The specified players will be looked up in the MusicalEnvironment.

```
pauseN e [″ sampler1″ ,″ sampler2″ ]
```

Those players could be restarted at a specified time, in this case the first beat of the 16th measure, using the playNAt function. The string after “e” is the specified time, given in terms of measure and beat.

```
playNAt e ″ 15.0″  [″ sampler1″ ,″ sampler2″ ]
```

The tempo of a particular TempoClock can be changed with the changeTempo function. The string “default” is the name of the TempoClock that is to be manipulated.

```
changeTempo e ″ default″  130
```

A new IOI function can be created. This function call gives the name “newIOI” to an IOI function which will be stored in the MusicalEnvironment. That string is followed by the offset, the number of beats before the first event takes place. The list contains IOI values; in this case, an interval of three beats passes between the first two events.

```
newIOIFunctionAndIOIList e ″ newIOI″
    0 [3,0.25,1,0.5,2,0.25,3]
```

A player can be told to use this new IOI function by calling the swapIOI function. After specifying the MusicalEnvironment, the name of the player and the name of the IOI function are given.

```
swapIOI e ″ sampler2″  ″ newIOIPattern″
```

All of the players can be stopped with the stopAll function.

```
stopAll e
```

8.2.2 Stochastic Rhythm Patterns Based on Sub-patterns

Pattern generation is based on a performer-selected core unit used to generate potential IOI values. Selection of a core unit, in conjunction with the length of the pattern, largely determines the metrical feel of the pattern. A list of scalars is determined by the performer, from which a function randomly selects a user-specified number of scalars.

The user specifies a number of subphrases to generate and the length of those phrases in terms of number of scalars to use, from which the final phrase will be constructed. Those subphrases are generated to the specified length by randomly choosing the specified number of scalars from the subset selected above and multiplying them by the core unit.

Finally, a user-specified number of subphrases are chosen at random from the resulting list by the algorithmic composition function. The user determines the length of the final phrase in terms of beats. If the length of the concatenated subphrases does not equal the specified length, the final IOI value is padded. If the length exceeds the specified length, the final IOI value is truncated.

An example of those steps follows. Items are determined by the user are followed with a "u":

- core unit (u): 0.25
- potential scalars (u): 1, 2, 3, 4, 5, 6, 7, 8
- number of scalars to be selected (u): 5
- selected scalars: 1, 2, 3, 4, 6
- potential IOIs: 0.25, 0.5, 0.75, 1.0, 1.5
- number of subphrases (u): 2
- subphrase length (u): 3
- selected subphrase scalars: 1, 3, 2; 4, 1, 6
- initial subphrases: 0.25, 0.75, 0.5; 2, 0.25, 2.5
- phrase length in terms of subphrases (u): 3
- initial randomly determined phrase: 0.25, 0.75, 0.5, 0.25, 0.75, 0.5, 2, 0.25, 2.5

- total phrase length: 7.75
- phrase length in terms of beats (u): 8
- final phrase: 0.25, 0.75, 0.5, 0.25, 0.75, 0.5, 2, 0.25, 2.75

8.2.3 An L-system Function for Conductive

The initial intention for using L-systems with the higher-order function described above is to generate the base rhythms from which the density table described above can be generated.

The module itself contains functions for generating a string output from an axiom, a rule set, and the generation number. Using the map function, a set of several generations can be obtained.

The rule set is notated with a colon rather than the traditional arrow for speed of entry. The predecessor symbol and successor string are written without spaces and separated by a colon. Each production rule must be separated by a space. The previous example can be run in ghci as follows:

```
*LSystem> let rules = ″ a:b b:ab″
*LSystem> getGeneration2 1 rules ″ a″
″ a″
*LSystem> getGeneration2 2 rules ″ a″
″ b″
*LSystem> getGeneration2 3 rules ″ a″
″ ab″
*LSystem> getGeneration2 4 rules ″ a″
```

```
" bab"

*LSystem> getGeneration2 5 rules " a"

" abbab"
```

A more complicated example follows:

```
rules: " a:ab b:acd d:gx e:abc f:ga g:d"

axiom: " abcdefg"
```

A symbol which has no rule is kept as-is. In is the same as if the rule were to repeat the symbol, such as “c -> c”.

In this case, the output in the interpreter of the first three generations of this L-system are:

```
*LSystem> getGeneration2 1 " a:ab b:acd d:gx e:abc f:ga g:d"  " abcdefg"

" abcdefg"

*LSystem> getGeneration2 2 " a:ab b:acd d:gx e:abc f:ga g:d"  " abcdefg"

" abacdcgxabcgad"

*LSystem> getGeneration2 3 " a:ab b:acd d:gx e:abc f:ga g:d"  " abcdefg"

" abacdabcgxcdxabacdcdabgx"
```

Two methods have been tested:

- direct output of IOI values
- lists of value-transforming functions applied in sequence to a base value

The direct output of IOI values means that given an axiom, a rule set, the generation number, and a list of potential IOI values, the function will return a list of IOI values. How those IOI values are assigned to the symbols is a matter for which a large variety of options exist. One simple choice is to randomly assign a value to each unique symbol. The string is then rewritten as that list of numeric values. This example illustrates such a method. The list of values ranges from 0.25 to 1.25, containing every step of 0.25.

```
*LSystem> getGeneration2 5 rules ″ a″
″ abbab″
*LSystem> let a = it
*LSystem> randomFinalizer2 [0.25,0.5..1.25] a
[1.25,0.25,0.25,1.25,0.25]
*LSystem> randomFinalizer2 [0.25,0.5..1.25] a
[1.25,1.0,1.0,1.25,1.0]
*LSystem> randomFinalizer2 [0.25,0.5..1.25] a
[0.5,0.75,0.75,0.5,0.75]
```

In the case of using transform, the rules of the L-system are mathematical functions that modify a numerical value. First the output of the L-system is similarly rewritten with one mathematical function randomly chosen for each unique symbol in the string. Given a starting value and that list of mathematical functions, the number is passed through the list so that the output of one function becomes the input of the next. The changes are accumulated so that each step in the transformation of the initial number is kept. That series of numbers is then processed as deltas on which the density function will generate variations. Here is

a very simple example of a list of functions processing a value:

```
*> transform 2 [(2 + ),((-1) +),(3 *)]

[2,4,3,9]
```

Here is an application of using the transform function on the output of an L-system.

```
*LSystem> getGeneration2 5 ″ a:b b:ab″

″ a″

″ abbab″

*LSystem> let a = it

*LSystem> let b = nub a

*LSystem> b

″ ab″

*LSystem> let c = zip (map (\x -> [x]) b) [(1+),(0.5*)]

*LSystem> let d = flatFinalizer c a

*LSystem> :t d

d :: [Double -> Double]

*LSystem> transform 2 d

[2.0,3.0,1.5,0.75,1.75,0.875]
```

In this example, the variable “d” is the output of the function flatFinalizer, which converts the symbols in a string to their equivalents in a dictionary. In this case, the dictionary is “c”, which maps the output of the L-system, “b”, to the list of operations above. The ghci command “:t” shows the type of something, and in this case is used to show that d is a list of

functions which take a Double and return a Double. The nub function returns a list in which all duplicate items have been removed. In this case, “d” would expand to:

```
*LSystem> transform 2 [(1+),(0.5*),(0.5*),(1+),(0.5*)]

[2.0,3.0,1.5,0.75,1.75,0.875]
```

The final output in both of these examples, i.e. the list of Doubles, is used as a list of IOI values. Those values are then processed into a density map as described in sections 2 and 3.1.

References

Aaron, Samuel, Alan F Blackwell, Richard Hoadley, and Tim Regan. “A Principled Approach to Developing New Languages for Live Coding.” In **Proceedings of New Interfaces for Musical Expression**, 381–86, 2011.

Abelson, Harold, and Gerald Jay Sussman. “Structure and Interpretation of Computer Programs,” 1983. http://mitpress.mit.edu/sicp/full-text/book/book.html.

Adriaensen, Fons. **Kokkini Zita - Linux Audio**, 2004. http://kokkinizita.linuxaudio.org/linuxaudio/.

Alperson, Philip. “On Musical Improvisation.” **The Journal of Aesthetics and Art Criticism** 43, no. 1 (1984): 17–29.

Anderson, David P., and Ron Kuivila. “Formula: A Programming Language for Expressive Computer Music.” **Computer** 24, no. 7 (July 1991): 12–21. doi:10.1109/2.84829.

Anderson, Elizabeth. “Dewey’s Moral Philosophy.” In **The Stanford Encyclopedia of Philosophy**, edited by Edward N. Zalta, Fall 2012., 2012. http://plato.stanford.edu/archives/fall2012/entries/dewey-moral/.

Ariza, Christopher. “Two Pioneering Projects from the Early History of Computer-Aided Algorithmic Composition.” **Computer Music Journal** 35, no. 3 (2011): 40–56.

Arnheim, Rudolf. **Visual Thinking**. Univ of California Press, 1969.

Attali, Jacques. **Noise: The Political Economy of Music**. Vol. 16. Manchester University Press, 1985.

Auslander, Philip. “Liveness, Mediation and Intermedial Performance.” **Degrés: Revue de**

Synthèse À Orientation Sémiologique, no. 101 (2000).

———. **Liveness: Performance in a Mediatized Culture**. Routledge, 2008.

Bailey, Derek. **Improvisation: Its Nature And Practice In Music**. Da Capo Press, 1993. http://www.worldcat.org/isbn/0306805286.

Bell, Paul. "Interrogating the Live: A DJ Perspective," 2010.

Bell, Renick. "An Interface for Realtime Music Using Interpreted Haskell." In **Proceedings of LAC 2011**. Maynooth, Ireland, 2011. http://lac.linuxaudio.org/2011/papers/35.pdf.

———. **Conductive-Base**, 2012. http://hackage.haskell.org/package/conductive-base.

———. "Considering Interaction in Live Coding Through a Pragmatic Aesthetic Theory." In **Proceedings of SI13, NTU/ADM Symposium on Sound and Interactivity**. Nanyang Technological University, Singapore, 2013.

———. **THE3RD2ND_011 - Renick Bell - Ragalike (Audio) (2007) [The3rd2nd.com – a Netlabel for Experimental Electronic Music]**, 2007. http://the3rd2nd.com/wiki/doku.php?id=the3rd2nd_011_-_renick_bell_-_ragalike.

———. "Towards Useful Aesthetic Evaluations of Live Coding." In **Proceedings of the International Computer Music Conference**. Perth, Australia, 2013.

Berg, Paul. "Abstracting the Future: The Search for Musical Constructs." **Computer Music Journal** 20, no. 3 (1996): 24–27.

Blackwell, AF, T Green, and Dje Nunn. "Cognitive Dimensions and Musical Notation Systems." **Workshop on Notation and Music Information Retrieval**, 2000. http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:Cognitive+Dimensions+and+Musical+Notation+Systems#0.

Blackwell, Alan, and Nick Collins. “The Programming Language as a Musical Instrument.” In **Proceedings of PPIG05**. University of Sussex, 2005.

Born, Georgina. “On Musical Mediation: Ontology, Technology and Creativity.” **Twentieth-Century Music** 2, no. 1 (2005): 7–36.

Bown, Oliver, Renick Bell, and Adam Parkinson. “Examining the Perception of Liveness and Activity in Laptop Music: Listeners’ Inference About What the Performer Is Doing from the Audio Alone.” In **Proceedings of the International Conference on New Interfaces for Musical Expression**. London, UK, 2014.

Brown, Andrew R. “Code Jamming.” **M/C: A Journal of Media and Culture** 9, no. 6 (2007).

Brown, Andrew R., and Andrew Sorensen. “Interacting with Generative Music Through Live Coding.” **Contemporary Music Review** 28, no. 1 (2009): 17–29. http://www.tandfonline.com/doi/abs/10.1080/07494460802663991.

Brown, Chris, and John Bischoff. “Indigenous to the Net: Early Network Music Bands in the San Francisco Bay Area,” 2002. http://crossfade.walkerart.org/brownbischoff/IndigenoustotheNetPrint.html.

Brown, Lee B. “‘Feeling My Way’: Jazz Improvisation and Its Vicissitudes-A Plea for Imperfection.” **Journal of Aesthetics and Art Criticism**, 2000, 113–23.

Clayton, Martin. “Time in Indian Music.” **Oxford Monographs on Music. Oxford University Press**, 2000.

Collins, Nick. “Generative Music and Laptop Performance.” **Contemporary Music Review** 22, no. 4 (2003): 67–79. doi:10.1080/0749446032000156919.

———. “Live Coding of Consequence.” **Leonardo** 44, no. 3 (2011): 207–11.

———. “Origins of Live Coding.” London, UK, March 2014. http://www.livecodenetwork.org/files/2014/05/originsoflivecoding.pdf.

Collins, Nick, and Alex McLean. “Algorave: A Survey of the History, Aesthetics and Technology of Live Performance of Algorithmic Electronic Dance Music.” In **Proceedings of the International Conference on New Interfaces for Musical Expression**, 2014.

Collins, Nick, Alex McLean, Julian Rohrhuber, and Adrian Ward. “Live Coding in Laptop Performance.” **Organised Sound** 8, no. 03 (2003): 321–30. doi:10.1017/S135577180300030X.

Cooper, Grosvenor. **The Rhythmic Structure of Music**. Vol. 118. University of Chicago Press, 1963.

Coutinho, C. “Tslime,” October 2010. http://www.vim.org/scripts/script.php?script_id=3023.

Cox, Geoff, Alex McLean, and Adrian Ward. “The Aesthetics of Generative Code.” In **Proc. of Generative Art**, 2001.

Davies, Stephen. **Musical Works and Performances: A Philosophical Exploration**. Oxford University Press, USA, 2001.

Davis, Paul. **Ardour**, 2006. http://ardour.org.

Deleuze, Gilles. “Lecture Transcripts on Spinoza’s Concept of Affect.” **Webdeleuze.com**, 1978. http://www.webdeleuze.com/php/texte.php?cle=14&groupe=Spinoza&langue=2.

Deleuze, Gilles, and Felix Guattari. **A Thousand Plateaus: Capitalism and Schizophrenia**. Burns & Oates, 1987.

Dewey, John. **Art as Experience**. Perigee Trade, 2005.

———. **Experience and Nature**. Vol. 1. DoverPublications. com, 1958.

———. “The Logic of Judgments of Practise.” **The Journal of Philosophy, Psychology and Scientific Methods**, 1915, 505–23.

———. “Theory of Valuation.” **International Encyclopedia of Unified Science**, 1939.

———. “Valuation and Experimental Knowledge.” **The Philosophical Review** 31, no. 4 (1922): 325–51.

Dexter, S, M Dolese, A Seidel, and A Kozbelt. “On the Embodied Aesthetics of Code.” **Culture Machine** 12, no. 0 (2011). http://www.culturemachine.net/index.php/cm/article/view/438/472.

Dijkstra, Edsger W. “E.W.Dijkstra Archive: A Short Introduction to the Art of Programming (EWD 316).” **E. W. Dijkstra Archive**, August 1971. http://www.cs.utexas.edu/~EWD/transcriptions/EWD03xx/EWD316.html.

———. “On the Role of Scientific Thought.” In **Selected Writings on Computing : A Personal Perspective**. New York: Springer, 1982. http://www.cs.utexas.edu/~EWD/transcriptions/EWD04xx/EWD447.html.

Dijkstra, Edsger Wybe, Edsger Wybe Dijkstra, and Edsger Wybe Dijkstra. **Notes on Structured Programming**. Technological University, Department of Mathematics, 1970.

Drape, Rohan. **Haskell Supercollider, a Tutorial**, 2009.

DuBois, Roger Luke. “Applications of Generative String-Substitution Systems in Computer Music.” PhD thesis, Columbia University, 2003.

Fishwick, Paul A. “On the Aesthetics of Programming and Modeling: Part 1: Evolving Program to Model,” 2000. http://citeseer.uark.edu:8380/citeseerx/viewdoc/summary?doi=10.1.1.28.4058.

Foltman, Krzysztof, Markus Schmidt, Christian Holschuh, and Thor Johansen. **Home @ Calf Studio Gear - Audio Plugins**, 2007. http://calf.sourceforge.net/.

Galanter, Philip. “What Is Generative Art? Complexity Theory as a Context for Art Theory.” In **In GA2003 – 6th Generative Art Conference**, 2003.

Gaspar, Alessio, and Sarah Langevin. “Restoring Coding with Intention in Introductory Programming Courses.” In **Proceedings of the 8th ACM SIGITE Conference on Information Technology Education**, 91–98, 2007.

Goehr, Lydia. **The Imaginary Museum of Musical Works: An Essay in the Philosophy of Music**. Oxford University Press, USA, 1994.

Green, Thomas R. “Cognitive Dimensions of Notations.” In **Proceedings of the Fifth Conference of the British Computer Society, Human-Computer Interaction Specialist Group on People and Computers V**, 443–60. New York, NY, USA: Cambridge University Press, 1989. http://www.cl.cam.ac.uk/users/afb21/CognitiveDimensions/papers/Green1989.pdf.

Green, Thomas, and Alan Blackwell. **Cognitive Dimensions of Information Artefacts: A Tutorial**, 1998. http://www.cl.cam.ac.uk/ afb21/CognitiveDimensions/ CDtutorial.pdf.

Gregg, Melissa, and Gregory J Seigworth. **The Affect Theory Reader**. Duke University Press, 2010.

Gresham-Lancaster, Scot. “The Aesthetics and History of the Hub: The Effects of Changing Technology on Network Computer Music.” **Leonardo Music Journal** 8 (1998): 39. http://www.jstor.org/discover/10.2307/1513398?uid=3738328&uid=2134&uid=4578002247&uid=2&uid=70&uid=3&uid=4578002237&uid=60&sid=21102875061697.

Harger, Honor. “Why Contemporary Art Fails to Come to Grips with Digital. A Response to Claire Bishop. « Honor Harger.” **Honor Harger - Musings on Art, Science, Particles & Waves**, September 2012. http://honorharger.wordpress.com/2012/09/02/why-contemporary-art-fails-to-come-to -grips-with-digital-a-response-to-claire-bishop/.

Harkins, H. James. **A Practical Guide to Patterns**. http://www.dewdrop-world.net/sc3/sym09/, 2009.

Harper, Douglas. “In Person.” Online dictionary. **Online Etymology Dictionary**, 2014. http://www.etymonline.com.

———. “Live.” Online dictionary. **Online Etymology Dictionary**, 2014. http://www.etymonline.com.

HarperCollins. “Instantiation.” Online dictionary. **Collins English Dictionary**, 2014. http://www.collinsdictionary.com/dictionary/english/instantiation?showCookiePolicy=true.

Hedges, Stephen A. “Dice Music in the Eighteenth Century.” **Music & Letters** 59, no. 2 (1978): 180–87.

Hilgard, Ernest R. “The Trilogy of Mind: Cognition, Affection, and Conation.” **Journal of the History of the Behavioral Sciences** 16, no. 2 (1980): 107–17.

Hookway, Christopher. “Lotze and the Classical Pragmatists.” **European Journal of Pragmatism and American Philosophy** 1, no. 1 (2009): 1–9.

Howe, Denis. “Abstraction.” Online dictionary. **The Free Online Dictionary of Computing**, 1985. http://foldoc.org/.

Hughes, John. “Why Functional Programming Matters.” **The Computer Journal** 32, no. 2 (1989): 98–107.

Hutton, Graham. **Programming in Haskell**. Cambridge [u.a.: Univ. Press, 2010.

Joel Spolsky. "The Law of Leaky Abstractions." **Joel on Software**, November 2002. http://www.joelonsoftware.com/articles/LeakyAbstractions.html.

Jones, Simon P., ed. **Haskell 98 Language and Libraries: The Revised Report**. http://haskell.org/, 2002. http://haskell.org/definition/haskell98-report.pdf.

Jones, SL Peyton, Cordy Hall, Kevin Hammond, Will Partain, and Philip Wadler. "The Glasgow Haskell Compiler: A Technical Overview." In **Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference**, Vol. 93, 1993.

Jr, Frederick P. Brooks. **The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition**. Anniversary. Addison-Wesley Professional, 1995.

Kaliakatsos-Papakostas, Maximos A, Andreas Floros, Nikolaos Kanellopoulos, and Michael N Vrahatis. "Genetic Evolution of L and FL-Systems for the Production of Rhythmic Sequences." In **Proceedings of the Fourteenth International Conference on Genetic and Evolutionary Computation Conference**, 461–68. ACM, 2012.

Kania, Andrew. "Pieces of Music: The Ontology of Classical, Rock, and Jazz Music," 2005.

Keller, RM. **Computer Science: Abstraction to Implementation**. Book manuscript in progress., 2001. www.cs.hmc.edu/ keller/cs60book/%20%20%20All.pdf.

Kitani, Kris M, and Hideki Koike. "Improvgenerator: Online Grammatical Induction for on-the-Fly Improvisation Accompaniment." In **Proceedings of the 2010 Conference on New Interfaces for Musical Expression (NIME 2010)**, 2010.

Knuth, Donald. "Computer Programming as an Art." **Paul Graham**, December 1974. http:

//www.paulgraham.com/knuth.html.

Kostelanetz, Ric. **Conversing with Cage**. Routledge, 2002.

Kozbelt, Aaron, Scott Dexter, Melissa Dolese, and Angelika Seidel. "The Aesthetics of Software Code: A Quantitative Exploration." **Psychology of Aesthetics, Creativity, and the Arts** 6, no. 1 (2012): 57.

Kramer, Jeff. "Is Abstraction the Key to Computing." **Communications of the ACM** 50 (2007).

Krueger, Charles W. "Software Reuse." **ACM Comput. Surv.** 24, no. 2 (June 1992): 131–83. doi:10.1145/130844.130856.

Kuivila, Ron. "A Performance at STEIM in 1985 in Which You Used the Formula Programming Language," November 2013.

Kuivila, Ron, The Hub, Julian Rohrhuber, Fabrice Mogini, Nick Collins, Volko Kamensky, Dave Griffiths, et al. "A Prehistory of Live Coding." TOPLAP, 2007. http://www.sussex.ac.uk/Users/nc81/toplap1.html.

Langston, Peter. "Six Techniques for Algorithmic Music Composition." In **15th International Computer Music Conference (ICMC), Columbus, Ohio, November**, 2–5. Citeseer, 1989.

Leddy, Tom. "Deweyś Aesthetics." In **The Stanford Encyclopedia of Philosophy**, edited by Edward N. Zalta, Fall 2012., 2012. http://plato.stanford.edu/archives/fall2012/entries/dewey-aesthetics/.

Leppert, Richard D. "Music 'Pushed to the Edge of Existence' (Adorno, Listening, and the Question of Hope)." **Cultural Critique** 60, no. 1 (2005): 92–133.

Liblit, Ben, Andrew Begel, and Eve Sweetser. "Cognitive Perspectives on the Role of Naming in Computer Programs." In **Proceedings of the 18th Annual Psychology of Programming Work-**

shop, 2006.

Lindenmayer, Aristid. “Mathematical Models for Cellular Interactions in Development I. Filaments with One-Sided Inputs.” **Journal of Theoretical Biology** 18, no. 3 (March 1968): 280–99. doi:10.1016/0022-5193(68)90079-9.

Liou, Cheng-Yuan, Tai-Hei Wu, and Chia-Ying Lee. “Modeling Complexity in Musical Rhythm.” **Complexity** 15, no. 4 (2010): 19–30.

Magnusson, Thor. “Confessions of a Live Coder.” In **Proceedings of International Computer Music Conference**, 2011.

———. “Epistemic Tools: The Phenomenology of Digital Musical Instruments,” 2009. http://eprints.brighton.ac.uk/6703/.

———. “The Threnoscope.” In **Proceedings of the 2013 International Conference on Software Engineering**, 2013.

Manousakis, Stelios. “Musical L-Systems.” **Koninklijk Conservatorium, The Hague (Master Thesis)**, 2006.

Marriott, Nicholas, and others. “Tmux,” January 2013. http://tmux.sourceforge.net/.

Massumi, Brian. “The Autonomy of Affect.” **Cultural Critique**, 1995, 83–109.

Maurer, John. **A Brief History of Algorithmic Composition**, 1999. https://ccrma.stanford.edu/blackrse/algorithm.html.

McCartney, J. “SuperCollider,” 1996. http://supercollider.github.io/.

Mccartney, James. “Rethinking the Computer Music Language: SuperCollider.” **Comput. Music J.** 26, no. 4 (2002): 61–68. doi:10.1162/014892602320991383.

McCormack, Jon, Oliver Bown, Alan Dorin, Jonathan McCabe, Gordon Monro, and Mitchell Whitelaw. “Ten Questions Concerning Generative Computer Art.” **Leonardo**, February 2013. doi:10.1162/LEON_a_00533.

McDermott, James, Jonathan Byrne, John Mark Swafford, Michael O’Neill, and Anthony Brabazon. “Higher-Order Functions in Aesthetic EC Encodings.” In **Evolutionary Computation (CEC), 2010 IEEE Congress on**, 1–8. IEEE, 2010.

McDougall, William. “Outline of Psychology.” 1923.

McLean, Alex. “Hacking Perl in Nightclubs,” 2004. http://www.perl.com/pub/a/2004/08/31/livecode.html.

———. “Tidal: Domain Specific Language for Live Coding of Pattern.” Documentation, 2014. https://github.com/yaxu/Tidal/blob/master/doc/tidal.md.

———. “TOPLAP Website.” **TOPLAP**, September 2010. http://www.toplap.org/index.php/Main_Page.

McLean, Alex, and Hester Reeve. “Live Notation: Acoustic Resonance?” In **Proceedings of the International Computer Music Conference**, 2012.

McLean, Alex, and Geraint Wiggins. “Live Coding Towards Computational Creativity.” In **Proceedings of the International Conference on Computational Creativity**, 175–79, 2010.

———. “Texture: Visual Notation for Live Coding of Pattern.” In **Proceedings of the International Computer Music Conference**, 2011.

———. “Tidal–Pattern Language for Live Coding of Music.” In **Proceedings of the 7th Sound and Music Computing Conference**, 2010.

McLean, Alex, Dave Griffiths, Nick Collins, and Geraint Wiggins. “Visualisation of Live Code.” **Proceedings of Electronic Visualisation and the Arts 2010**, 2010.

McLean, Alex, Thor Magnusson, and Nick Collins. “DVD Program Notes.” **Computer Music Journal** 35, no. Winter 2011, No. 4 (December 2011): 119–37.

McLean, Alex, Adrian Ward, and others. “LivecodingGrades.” **TOPLAP**, July 2005. http://toplap.org/wiki/LivecodingGrades.

Moolenaar, Bram. **The Vim Editor**, 2008. http://www.vim.org.

Morgan, Nigel. “Transformation and Mapping of L-Systems Data in the Composition of a Large-Scale Instrumental Work.” In **Proceedings of ECAL 2007 Workshop on Music and Artificial Life (MusicAL 2007)**, 2007.

Nielsen, Jakob. “10 Usability Heuristics for User Interface Design.” **Nielsen Norman Group**, January 1995. http://www.nngroup.com/articles/ten-usability-heuristics/.

Nilson, Click. “Live Coding Practice.” In **NIME ’07: Proceedings of the 7th International Conference on New Interfaces for Musical Expression**, 112–17. New York, NY, USA: ACM, 2007. doi:10.1145/1279740.1279760.

Nishino, Hiroki. “Cognitive Issues in Computer Music Programming.” In **Proceedings of the International Conference on New Interfaces for Musical Expression**, 499–502. Oslo, Norway, 2011.

Norman, Donald A. “Categorization of Action Slips.” **Psychological Review** 88, no. 1 (1981): 1–15. doi:10.1037/0033-295X.88.1.1.

Nyman, Michael. **Experimental Music: Cage and Beyond (Music in the Twentieth Century)**. 2nd ed. Cambridge University Press, 1999. http://www.worldcat.org/isbn/0521653835.

O'Sullivan, Stephen, and Pecorino, Philip. "Ethics - an Online Textbook," 2002. http://www2.sunysuffolk.edu/pecorip/SCCCWEB/ETEXTS/ETHICS/.

Paine, Garth, and Jon Drummond. "Developing an Ontology of New Interfaces for Realtime Electronic Music Performance." **Proceedings of the Electroacoustic Music Studies (EMS)**, 2009.

Parncutt, Richard. "A Perceptual Model of Pulse Salience and Metrical Accent in Musical Rhythms." **Music Perception: An Interdisciplinary Journal** 11, no. 4 (1994). http://dx.doi.org/10.2307/40285633.

Paul, David. "Karlheinz Stockhausen." **Interview, Seconds Magazine** 44 (1997).

Piñeiro, Erik. "Instrumentality : A Note on the Aesthetics of Programming." **Consumption Markets & Culture** 5, no. 1 (2002): 63–68. doi:10.1080/1025386029003118.

Predelli, Stefano. "Has the Ontology of Music Rested on a Mistake?" **Literature & Aesthetics** 12 (2011).

Princeton, University. "About WordNet." **WordNet**, 2010. http://wordnet.princeton.edu/.

Robillard, David. **Patchage**, 2011. http://drobilla.net/software/patchage/.

Rodman, Selden. **Conversations with Artists**. Devin-Adair Co., 1957.

Rorty, Richard. **Consequences of Pragmatism: Essays, 1972-1980**. U of Minnesota Press, 1982.

Russell, James A. "A Circumplex Model of Affect." **Journal of Personality and Social Psychology** 39, no. 6 (1980): 1161.

———. "Core Affect and the Psychological Construction of Emotion." **Psychological Review** 110, no. 1 (2003): 145.

———. “From a Psychological Constructionist Perspective.” In **Categorical Versus Dimensional Models of Affect: A Seminar on the Theories of Panksepp and Russell**, 2012.

Sawyer, R Keith. “Improvisation and the Creative Process: Dewey, Collingwood, and the Aesthetics of Spontaneity.” **The Journal of Aesthetics and Art Criticism** 58, no. 2 (2000): 149–61.

Schoenberg, Arnold. **Fundamentals of Musical Composition**. Edited by Gerald Strang and Leonard Stein. Faber & Faber, 1999.

Sethares, William A, and Diego Bañuelos. **Rhythm and Transforms**. Vol. 1. Springer, 2007.

Shaw, M. “Larger Scale Systems Require Higher-Level Abstractions.” **SIGSOFT Softw. Eng. Notes** 14, no. 3 (April 1989): 143–46. doi:10.1145/75200.75222.

Shaw, Mary. “Toward Higher-Level Abstractions for Software Systems.” **Data & Knowledge Engineering** 5, no. 2 (July 1990): 119–28. doi:10.1016/0169-023X(90)90008-2.

Shelley, James. “The Concept of the Aesthetic.” In **The Stanford Encyclopedia of Philosophy**, edited by Edward N. Zalta, Spring 2012., 2012. http://plato.stanford.edu/archives/spr2012/entries/aesthetic-concept/.

Shusterman, Richard. **Performing Live: Aesthetic Alternatives for the Ends of Art**. Cornell University Press, 2000.

———. “The End of Aesthetic Experience.” **Journal of Aesthetics and Art Criticism** 55 (1997): 29–42.

Shutt, John N., and John N. Shutt. “Abstraction in Programming - Working Definition,” 1999. http://citeseer.uark.edu:8380/citeseerx/viewdoc/summary?doi=10.1.1.40.1415.

Simonyi, Charles. “Intentional Programming: Innovation in the Legacy Age.” In **IFIP Working Group**, 2:1024–43, 1996.

Smith, John Miles, and Diane C. P. Smith. “Database Abstractions: Aggregation and Generalization.” **ACM Trans. Database Syst.** 2, no. 2 (June 1977): 105–33. http://doi.acm.org/10.1145/320544.320546.

Soddu, Celestino. “Generative Art International Conference,” 2013. http://www.generativeart.com/.

Sorensen, Andrew. “A Study In Keith,” 2008. http://vimeo.com/2433947.

———. “Impromptu: An Interactive Programming Environment for Composition and Performance.” In **Proceedings of the Australasian Computer Music Conference 2009**, 2005.

Sorensen, Andrew, and Andrew R. Brown. “Aa-Cell in Practice: An Approach to Musical Live Coding.” In **Proceedings of the International Computer Music Conference**, 2007.

Spinoza, B de. **Ethics (Trans. RHM Elwes)**, 2007. http://www.gutenberg.org/ebooks/3800.

Supper, Martin. “A Few Remarks on Algorithmic Composition.” **Computer Music Journal** 25, no. 1 (2001): 48–53.

Swift, Ben. “Playing an Instrument (Part II).” Documentation. **Extempore Docs**, 2014. http://benswift.me/2012/10/15/playing-an-instrument-part-ii/.

Sylvan, Sebastian, Amir, and David Wahler. “Why Haskell Matters - HaskellWiki,” January 2012. http://www.haskell.org/haskellwiki/Why_Haskell_Matters.

Thielemann, Henning. “Live Music Programming in Haskell.” **ArXiv Preprint ArXiv:1303.5768**, 2013.

Tingen, Paul. “Autechre, Recording Electronica.” **Sound on Sound** 19, no. 6 (2004): 96–102.

Toussaint, Godfried T. **The Geometry of Musical Rhythm: What Makes a“ Good” Rhythm Good?** CRC Press, 2013.

Turner, Raymond, and Amnon Eden. “The Philosophy of Computer Science.” In **The Stanford Encyclopedia of Philosophy**, edited by Edward N. Zalta, Winter 2011., 2011. http://plato.stanford.edu/archives/win2011/entries/computer-science/.

Van Leeuwen, Jan. “Toward a Philosophy of Information and Computing Sciences.” **Netherlands Institute for Advanced Study Newsletter**, no. 42 (2009): 22–25.

various. **Haskell Hierarchical Libraries**, 2013. http://www.haskell.org/ghc/docs/latest/html/libraries/index.html.

Varouhakis, John, and Martin Nordholts. **recordMyDesktop Version 0.3. 7.3**, 2008.

Wang, Ge, and Perry Cook. “ChucK: A Programming Language for on-the-Fly, Real-Time Audio Synthesis and Multimedia.” In **MULTIMEDIA ’04: Proceedings of the 12th Annual ACM International Conference on Multimedia**, 812–15. New York, NY, USA: ACM, 2004. http://dx.doi.org/10.1145/1027527.1027716.

Ward, Adrian, Julian Rohrhuber, Fredrik Olofsson, Alex McLean, Dave Griffiths, Nick Collins, and Amy Alexander. “Live Algorithm Programming and a Temporary Organisation for Its Promotion.” In **Proceedings of the README Software Art Conference**, 2004.

West, Cornel. **The American Evasion of Philosophy: A Genealogy of Pragmatism**. Univ of Wisconsin Press, 1989.

Wishart, Trevor. **On Sonic Art**. Vol. 12. Routledge, 1996.

Worth, Peter, and Susan Stepney. “Growing Music: Musical Interpretations of L-Systems.” In **Applications of Evolutionary Computing**, 545–50. Springer, 2005.

Wright, Peter, and John McCarthy. **Technology as Experience**. MIT Press, 2004.

Xenakis, Iannis. **Formalized Music: Thought and Mathematics in Composition**. 2nd ed. Pendragon Pr, 2001.

Zachar, Peter, and Ralph D Ellis. **Categorical Versus Dimensional Models of Affect: A Seminar on the Theories of Panksepp and Russell**. Vol. 7. John Benjamins Publishing, 2012.

Zmoelnig, IOhannes M. “Pointillism,” December 2012. http://umlaeute.mur.at/Members/zmoelnig/projects/pointillism/.