SOFTWARE MUSIC

A Thesis

Submitted to the Faculty

in partial fulfillment of the requirements for the

degree of

Master of Arts

in

Digital Musics

by

Michael Chinen

DARTMOUTH COLLEGE

Hanover, New Hampshire

May 30, 2009

Examining Committee:

_____

(chair) Michael Casey

_____

Charles Dodge

_____

Richard Karpen

_____

Andrew Campbell

_____

Brian W. Pogue, Ph. D.

Dean of Graduate Studies

Abstract

Software is an essential component of computer music. Despite parallel trends toward composing in a graphical environment, current trends in computer music still offer composers the option to write code. Both graphical music composition environments and computer music programming languages tend to hide the software elements from the composer by modeling instrument/score composition paradigms. At the same time, other fields concerned with aesthetics, such as design and media art, are exhibiting trends toward 'post-graphical user interface' paradigms using programming.

This thesis shows how it is possible to use programming concepts such as reference, hierarchy, and synchronicity to directly create musical reference, musical hierarchy, and musical synchronicity. Its aim is to make more transparent the relationship between programming and the music it creates. These concepts are also present in natural systems, and their use as such in artistic material should be considered. It presents several compositions written by the author using these concepts: *Tree Music*, explores hierarchy and reference among data structures to provide hierarchical interpolation. *Limited Resources* explores synchronicity and threads in the software context by relating them to an ecosystem. *Dict*, explores reference and hierarchy using online dictionary definitions to deconstruct the meaning of a sentence.

Preface

This thesis contains my insights about music when writing software, as well as insights about software when writing music. These insights have led to several software music pieces. As such, I hope that this thesis will provide a means to understand these works.

The main thrust of the thesis is to draw connections between musical concepts and those in computer science to better understand what it means to write music in a programming language. Perhaps more importantly, it shows in some detail how programming relates to music, art, and nature. Working with data structures and algorithms in the study of computer science, and especially in the practice of programming real-world problems, I have come to view data structures as having elegance and complexity. I have applied these directly to help create musical structure and form. Also, many non-musical computer programs have complex systems that exhibit behaviors that are analogous to both natural systems and music.

There are concepts in this thesis from, and related to, programming, but fluency in a programming language is not necessary for reading this thesis. The concepts will be presented in English, with pictures, and with metaphors because I believe these concepts transcend the code in which they are implemented. The connections to ideas outside the computing world are more obvious when they are discussed in this way. A vast amount of music

has been written using computer programming. Much of it does not take full advantage of certain powerful concepts used in general computer programming. In the past few years as a programmer I found myself thinking about programming in a very different way when I approached a musical application. I noticed that the structure and code flow in the musical software I was writing tended to be more straightforward, simpler, and less hierarchical than the multithreaded, networked code I was writing for my computer software purposes. I found this problematical, because the resultant piece's musical structure had a close relationship with the less elaborate program structure. This led me to begin to ask questions about the amount of influence the history of the available software tools were having on my work as a computer music programmer. As such, this thesis is written with respect to the history of computer music, and my hope is that this will make it useful to other computer musicians.

Acknowledgements

This thesis would not have been conceived if it were not for the helpful support and engaging company of the faculty, students, and staff at Dartmouth College's Digital Musics program.

I would like to thank my advisors, Michael Casey, Charles Dodge, Richard Karpen, and Andrew Campbell. Each provided a unique perspective that allowed me to step out of my box. Additional thanks goes out to my advisors for a painstaking detail to editing my writing, draft after draft.

All of the pieces in this thesis have themes that are present in works by my student peers at Dartmouth. While new music often focuses on what is different, this local similarity deserves acknowledgement and celebration. To me it means that I have been in a good community – we have listened to one another beyond superficial discussion.

Special thanks goes out to Chris Peck and Paul Osetinsky for improving my writing, and Kristina Wolfe for reviewing my presentation. I would also like to thank Jon Appleton, Newton Armstrong, Kui Dong, Larry Polansky, Hiroki Nishino, Naotoshi Osaka, Spencer Topel, Rebecca Fawcett, John Arroyo, Patrick Barter, Beau Sievers, Wally Smith, and Barbara Smith for their support. Finally, thanks to my family and friends for being there, and for reminding me of my existence outside of writing this thesis.

Table of Contents

List of Figures

1 INTRODUCTION

Despite its cryptical reputation, computer programming is about humans. It is about their needs and desires, and therefore, it is about their nature. Humans, not computers, write software that other humans use to achieve their goals. The cryptical aspect of programming comes from the level of technical mastery required to effectively engage in it. As in the creation of art, programming requires technical mastery necessitating control over the material at many levels [1], [2]. Computer music from the 1950's onwards has been written using means that either include programming, or have been influenced by it. Musical programming languages, as will be shown, have exhibited a trend towards hiding the low-level programming from the composer. Depending upon the composer's goals, this may not pose a problem. However, low-level details may in fact hold conceptual significance for the composer, such as the analogies the software's system may have to natural, human, or musical behavior.

Computer programming, or coding, for the scope of this thesis, will refer to the process of producing and testing of text-based source code to be compiled

into programs. This definition excludes the use of graphical development environments such as *Pd* [3] or *Max/MSP* [4], which, while having programming-like elements, do not easily facilitate the implementation of common programming concepts, such as recursion or polymorphism. The distinction between high and low level programming is frequently referred to, so it will be useful to provide a definition. Low-level programming, and low-level programming languages have less abstraction from the instructions that the processor reads and executes. High-level programming attempts to remove the implementation details specific to computers, using abstractions such as objects instead of data structures, variables instead of registers, and functions that download data from the internet without requiring the programmer to explicitly open a TCP/IP socket.

It should be noted that 'high-level' and 'low-level' are relative terms here, as C is a high-level programming language with comparison to assembly, since it encapsulates data structures, but a low-level language compared to Java or Perl, as it allows one to manipulate physical memory addresses. It should also be pointed out that certain languages allow convenient access to multiple levels of programming. Objective-C++, for example, is backwards-compatible with most C and C++ programming, while it adds an additional object-oriented level, in which C or C++ code can be embedded within. For the scope of this thesis the term “low-level” will refer to languages such as C/C++, and

"high-level" will be used relative to music programming languages such as *SuperCollider*.

In the early days of computer music, when the only well-known music-specific creation tool for computers was Mathews's *MUSIC*-N series [5], programming was a prerequisite to creating a piece of music using a computer. The *MUSIC*-N programs have an instrument file, which defines the signal processing methods (unit generators) that an instrument file can use. The instrument file defines how the unit generators are used throughout the piece, by specifying parameters such as start time, duration, pitch, amplitude, and so forth. *MUSIC-N*'s concept of the unit generator has had a lasting influence on the design of subsequent music programming languages, such as CSound[6], CMix[7], and CMusic[8].

Today composers can create music with tools that use a graphical development environment, such as *Max/MSP* or *Pd*. By contrast, *SuperCollider [9]* and ChucK[10] represent another trend in current tools to further expand high-level coding. In some cases, such as in the practice known as 'live coding' [11], the very act of writing code becomes an important part of the piece. Whether to program or not has become an important choice for today's computer musicians.

The trend towards high-level programming also exists outside of music. Currently a number of higher level languages are in active use, including so

called 'scripting' languages such as Ruby and Python. However, unlike the musical programming languages mentioned above, these coexist with lower level programming languages, such as C or Java, so that the programmer can choose the levels at which he wants to work.

The graphical development paradigms in Max/MSP are reflected in development environments for C++, Java, or Objective-C. These lower level programming languages are often paired with tools that, although not a part of the language, allow for a graphical editing of certain parts of the code that are graphical in nature, such as the design of the graphical user interface (GUI.) This suggests that there are some aspects of programming that are facilitated by non-textual manipulation. However, these graphical editing tools ultimately insert text into the source files and modify code. The resulting code may be inspected and modified through text-based programming, so its use is optional and as such does not inhibit a text-based programmer. This stands in contrast to *Max/MSP*, which does not provide a source-code counterpart.

There are recent trends towards a post-GUI aesthetic, especially in media and software art, which recognizes the importance of the programming side of software [12], [13], [14]. These trends were influenced by the study of design, including Christopher Alexander's work in recognizing abstracted patterns in architectural design of multiple scales, from windows to towns, as

well as the relationships between the levels [15]. One result of this trend is the popular *Processing* programming language for real time applications, based on the book *Design By Numbers[16]* by John Maeda. Florian Cramer and others have also worked on the idea of 'Software Art' [13], an approach which engages its audience with work that goes beyond the GUI through making the underlying code more transparent.

While researchers in the fields of music information retrieval and sound analysis/synthesis often use complex data structures and algorithms, these topics are discussed much less frequently for compositional use. One possible explanation for this is that composers who come from traditional score-based composition backgrounds have found it more intuitive to use software to model scores and instruments. This approach hides some of the computer programming from the user. Furthermore, not all composers that write code are familiar with computer science, as computer science topics are not a part of standard music composition pedagogy. Even for the composers familiar with computer science, the music software they use can make implementing a data structure or object-oriented design cumbersome. For this reason, it may be helpful to reexamine programming languages to determine in what ways computer science tools can be useful for composing. The concepts of structure, hierarchy, reference, and synchronicity are commonly encountered in the classical music tradition, including computer music, so it seems that

their software counterparts might prove useful as compositional tools.

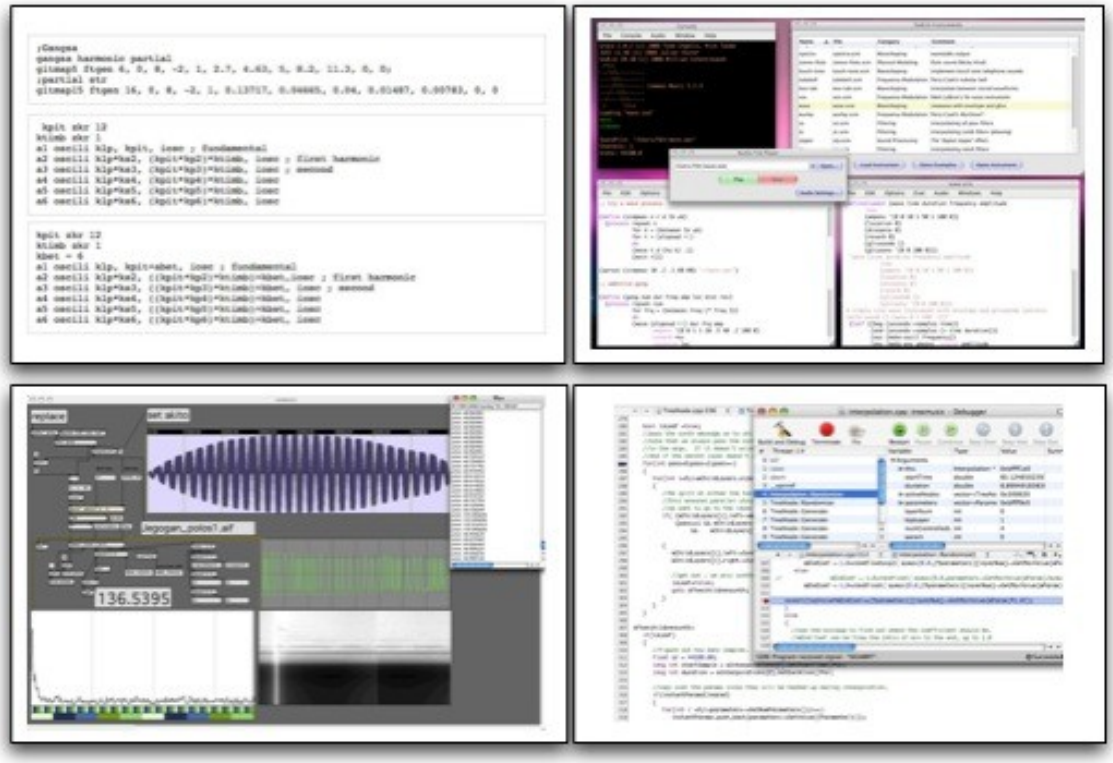

Figure 1. Four development environments for computer music. Csound (top left) and Common Lisp Music (top right) are related to Mathew's *Music* V program in many ways, including their instrument/score models. Max/MSP (bottom left) enables the composer to bypass text-based programming, using a set of precompiled unit-generators called 'externals.' XCode (bottom-right) is an integrated development environment that allows C++, Objective-C, and Java compiling and debugging for general applications. The XCode project shown here is a piece of music written by the author.

Some computer music pieces use data structures and object-oriented programming (OOP) to implement a musical structure and form [17], [18]. Often, the composer does not consider the data structure to be responsible for the inspiration of the musical structure; the composer simply feels that the data structure can create an effective model of the original musical idea. Similarly, data structures and object-oriented programming are almost

always used to model a problem or phenomenon. It might be said that the complexity of the data structures are merely a byproduct of the original problem's inherent structure. This puts some importance on the computer science or engineering practice of "defining the problem" in specific cases of computer music programming. For the author's compositions discussed in chapter 4, the design of the program flow and data structure is fundamental to the piece. Here the problem is to create an algorithm designed for the purpose of sonifying the structure of the data. Additionally, the process of developing and extending an algorithm yields new insights into the original idea.

In Charles Dodge's *Earth's Magnetic Field* (1970), geophysical data representing states of the earth's magnetic field over time is reinterpreted to form musical notes. *Earth's Magnetic Field* is the first example of so called 'data sonification.' Composers such as Kim Cascone and Yasunao Tone have written music that has the computer read non-audio data, such as a text file or JPEG image, as if it were an audio file, producing relatively noisy sounds that are structured by the changes in the file [19]. This has been recently termed 'data bending' [20]. Data bending differs from 'data sonfication' in that the intent of the latter is to transmit the original data through an acoustic encoding that (usually a human) can decode aurally, whereas the former has interest in misinterpretation. Composers from Dufay to Bartok to

Ryoji Ikeda have written music based around culturally significant numbers (e.g. the golden mean) and data (e.g. NYSE ticker data.) The data sonification practice is closer to the pieces presented in chapter 4 than data bending in that the sonification contains more information about the original data, but differs from 'data sonification' in that the source data is created and modified for optimal sonificaton.

To demonstrate the compositional ideas presented in this paper, several pieces have been composed using the C++ language. The C++ language was chosen because it is a well-known language, and it allows easy implementation of data structures and low-level programming concepts. Also, unlike music-specific programming languages, good debugging tools have evolved for C++, which are an enormous help when writing complex programs. The source code for the compositions in question is available online, but for the sake of a general audience, the pieces will be explained in English and pseudocode.

This thesis is organized as follows. Chapter 2 is a survey of related work is presented with respect to this thesis. Chapter 3 is a section on the concepts in data structures, design patterns, and object-oriented programming that are useful as compositional tools. Methods of programming and analyzing music composed with these concepts will be presented as well. In chapter 4, the implementation and results of several of my own compositions created

using the concepts from the previous chapter. Lastly, conclusions and future works are considered.

2 SURVEY OF RELATED WORK

2.1 Historical Overview

2.1.1 Fundamentals in Computer Science

Oram and many authors of the collection of articles by recognized programmers in *Beautiful Code [21]* describe programming using words such as 'elegant' and 'beautiful'. Indeed, the word 'art' is linked to the early days of computer science, as used in Donald Knuth's seminal work from the 1960's and 1970's, *The Art of Computer Programming [22]*. Given the context, one could argue that these words were being used simply to convey an appreciation of technical mastery. Even with such a utilitarian view of programming, the reality is that the applications of programming are wide-ranging enough that they include aesthetics. For example, Knuth was concerned with using computers and algorithms to advance typesetting [23]. Knuth felt there was a problem with the characters on a page being 'too perfect,' leading to his findings that adding slight randomized jitter to the pen positions produced a more pleasing result. Finally, it should be noted that Knuth has written and given a lecture for the purpose of explicitly stating that he was using the word 'art' in his title because he believed programming can be as artistic an endeavor as making a sculpture or poetry [24].

2.1.2 Object Oriented Programming and Design Patterns

As programming problems grew more complex, computer scientists found the need to create higher-level languages on top of the Turing-complete assembly language that, while flexible, was not efficient for humans to write and read. Although data structures could be implemented in assembly language, (e.g. the way Knuth chose to write his programming examples,) languages that made these structures more obvious to the eye became desirable. The invention of languages such as ALGOL, Fortran, and Pascal produced code, (and importantly, introduced a culture of spacing, naming, and other stylistic conventions [25],) which enabled a degree of structural comprehension at a glance. At the same time, programmers found themselves writing more complicated data structures and algorithms that interacted in ways similar to the interactions of physical objects [26]. This led to 'pure' object oriented languages such as Smalltalk. Eventually, certain existing languages, such as C and Lisp, which were not object oriented, were further developed to support object-oriented concepts, resulting in languages such as C++ and Common Lisp.

Programmers in the 80's and 90's found themselves writing similar data structures that could be used across many implementations. The publication of Christopher Alexander's *Notes on the Synthesis of Form* (1964) and *A Pattern Language* (1977) [27], (which were enormously influential with computer scientists,) led to the formalization of programming libraries, such

as the Standard Template Library (STL,) and design patterns in *Design Patterns: Elements of Reusable Object-Oriented Software* by the gang of four [28]. Examples of a pattern in architecture such as "Eccentric nucleus," "Windows overlooking life," and "Bus stop," abstract general elements of rooms, houses, or towns. In programming a design pattern examples include the Factory pattern, which creates and recycles objects dynamically, or the Singleton, a global state-tracking or managing pattern which only has one instance in existence, accessible from any part of the code.

2.1.3 Early Algorithmic Music

Formalization in music has a long history. While its ancient origins are unclear, (for that would depend on one's personal definition of a formal system or algorithm,) we can find a well-documented and interesting example from nearly a millennium ago [29]. Guido d'Arrezo, the Benedictine monk and musical theorist to whom developments in four-line staff notation are generally attributed, also presented an algorithmic system for selecting pitches when writing chants. The system required the composer to consult a lookup table while setting the text. The table dictated which pitches could be used for a given syllable based on its vowel.

There are many other historical rule-based systems in music. Some examples are Fux's species counterpoint, the canon, the fugue, Mozart's *Musikalisches Wiirfelspiel,* and sonata-allegro form. Current developments

in the use of complex formalized systems and AI have caused a number of computer musicians, such as Roads [30] and Loy [29] to take another look at some of these earlier systems. This research raises the question of where the "creativity" of the composer lies between him and the rule-based system. The early works in algorithmic composition provide insight for the philosophical questions the works raise about creativity and rule-based composition.

In Guido's case, the system could be used by one composer to create an entirely different piece compared to the one another might compose with the same text. Thus, it can be said that the system imposes compositional constraints on the composer, who remains free to impose his own musical inflections upon the music. It may be possible, and even tempting in the context of discussing algorithmic composition, to analyze all aspects of all composition as being rule-based. However, one must realize that if certain musical decisions do not come from explicit or easily recognizable rules, this may be a difficult exercise that does not provide much insight. Imposing rules on the 'less formal' components of music can easily lead to overfitting, (which will be discussed in Chapter 3.3). Music theory provides a rule-based analysis of western tonal music. In the development of tonal music, it has been important when composers have chosen to "break" the rules of the system. The early systems such as Guido's are interesting, because in works produced using them, the separation of rule-based and non-rule-based

components is easy to perceive.

2.1.4 Twelve-Tone Music and Serialism

Schoenberg's twelve-tone system [31] shares common elements with Guido's system: composers who use it let pitch be determined, to some degree, by the system. Twelve-tone technique places all twelve pitch classes in a row to be played sequentially in its entirety. Composers still have choice of pitch in the potential row transformations used. Transposition, inversion, and retrogression of the row forms increases the palette of choices. It is also similar to Guido's system in that it is a general system that any number of composers can use to write any number of pieces. Composers have chosen to use the system in a wide variety of ways and not just as a generator. For example, Webern's *Symphony* (op. 21) explores registral symmetry as well as symmetry in the matrix of transformations on the 12-tone row [32]. The first movement of op. 21 is also a double cannon. Webern was interested in working with two completely different systems that each had their own set of constraints to create structure in his music. The problem of solving for multiple constraints is a central problem for design, (discussed in 2.3.1).

Integral serialism is an extension of Schoenberg's twelve-tone technique from pitch onto other musical parameters. From a formal systems perspective, it is interesting to see the original system extend this way – it is perhaps evidence that in seeking out new compositional ideas, composers will

try to recycle not only musical content, but the systemic design as well. This is similar to the abstraction and reuse of patterns in architecture and programming. As will be shown in the recent developments section, this trend will increase as computers provide new systems not originally designed for use in music.

2.1.5 Process and Chance Music

A number of composers in the 1960's became interested in other techniques of writing music that did not focus on pitch as a central element. One relevant subgroup of these composers are those that wrote what has come to be known as process music. One well-known example of this is Alvin Lucier's *I Am Sitting in a Room,* where a speech recording is fed repeatedly from loudspeaker to microphone to tape recorder, amplifying the room resonances until by the nth generation, the speech message on the tape is unintelligible.

Process music introduces the concept of not only composing music with an algorithm, but also composing music to sonify the algorithm. In the motivation for process music lies the belief that there is something aesthetically interesting about processes [33]. This presents another musical motivation for writing music with computer programs, since algorithms and computer programs are always comprised of processes.

John Cage wrote music that has a relationship to process music. Beginning in the early 1950's, he wrote pieces that used the *I Ching* to make

decisions. It is important to note that although these are aleatoric decisions, Cage always decided in advance what aspects of the music were determined by chance. Thus it could be said that Cage developed processes that created structure around chance. Cage and Christian Wolff also wrote music that introduced elements of chance by explicitly asking the performers to make decisions. Examples of this are Cage's *Cartridge Music* and Wolff's contrasting *Music for 1, 2, or 3 Musicians*. These pieces make the process element more complex by turning the performers into observable processes whose actions could affect the processes of other performers. *In Cartridge Music*, performers make their own decisions as to exactly when a sound is made within a certain time span. In *Music for 1, 2, or 3 Musicians*, the performers are given performance instructions that are dependent and conditional upon the sounds made during the performance. Because the individual performers must listen to each other to determine their next instruction, it can be more explicitly seen as individual processes that interact with each other. This idea might be compared to multithreaded programming, which also is about simultaneous processes that have certain dependancies and constraints with each other.

2.1.6 Stochastic Music

In *The Crisis of Serial Music* [34], Iannis Xenakis presents a sharp critique of integral serialism. In particular, Xenakis makes the claim that such music

becomes so complex that the main thing that becomes perceivable are statistical elements of the music:

> Linear polyphony destroys itself by its very complexity; what one hears is in reality nothing but a mass of notes in various registers. The enormous complexity prevents the audience from following the intertwining of the lines and has as its macroscopic effect an irrational and fortuitous dispersion of sounds over the whole extent of the sonic spectrum. There is consequently a contradiction between the polyphonic linear system and the heard result, which is the surface or mass. This contradiction inherent in polyphony will disappear when the independence of sounds is total. In fact, when linear combinations and their polyphonic superpositions no longer operate, what will count will be the statistical mean of isolated states and of transformations of sonic components at a given moment. The macroscopic effect can then be controlled by the mean of the movements of elements which we select. The result is the introduction of the notion of probability, which implies, in this particular case, combinatory calculus. Here, in a few words, is the possible escape route from the "linear category" in musical thought. [35]

In this text, Xenakis hints at the idea that some perceptual parameters are not independent of others, which if true, certainly complicates the matter of systems such as integral serialism that treat parameters as totally separable. If it is believed that humans perceive statistically, Xenakis's use of statistics implies that there might be meaningful parameters that can be extracted from other parameters and values. This suggests that a relationship exists between parameters and hierarchy.

Xenakis distinguished himself by composing pieces that took advantage of statistics by means of stochastics. In order to generate the large volume of random numbers necessary to implement a stochastic distribution, he used a computer. Some of these pieces, such as *ST-10* have implementations that are well documented, with source code [36].

Although Xenakis now had a means of controlling statistical elements, he

still was faced with the problem of mapping his statistics to music. His programs were actually quite complex, with definite hierarchical structure that is stochastically, but explicitly, defined in his software for section, subsection, gesture, and so on. In other words, the structure of the piece can be seen in the program code, without looking at the stochastic elements. The choice to implement things in this hierarchical fashion is significant in that it establishes common ground between his philosophy and the classical tradition (note, motif, phrase, section,) including Schoenberg [37]. Xenakis does not talk about these decisions much, but the 'thru-composed' and intuitive hierarchy is at least as important as Xenakis's decision to use statistical distributions in the resulting piece.

2.2 Recent Musical Developments

2.2.1 AI, Dynamical Systems, and Biologically-Inspired Systems

There are many other musical works that use programming in a way that goes beyond the score/instrument model. One example is the large amount of music that draws upon AI techniques that originally had applications in general computer science, such as in simulation or search. Music written with the aid of genetic algorithms (GAs,) for example, became quite popular, [38], [39], [40]. However, bringing these algorithms into art is doubly complicated. The task of implementation aside, the composer must deal, (or decide not to deal,) with the relationship the piece will have to the algorithm's

philosophical references (e.g. in the case of GAs, Darwinism.) Also, the composer may choose to 'hack' the algorithm to do something, or use parts of it (e.g. deciding to use the 'less fit' chromosomes in a GA) that wouldn't be used in a software engineering context. The idea of writing music using AI in general comes with the enormous themes the nature of creativity and even life itself [41].

These ideas in AI [42], [43] use algorithms that operate at such a high level that the programming details are not very important to the philosophy and end product of the music. This is perhaps the main difference to the work presented in chapter 4, although there are other important differences and similarities. For the interested reader, in section 4.2., a piece written by the author is described and compared to an ecological system.

David Dunn's work takes a radically different approach to nature-inspired music [44]. It deals directly with the dynamical systems that have evolved in nature. In a work entitled *Mimus Polyglottos,* he presents a wild mockingbird in a tree with prerecorded synthesized tape music. The bird attempts to learn the alien electronic sounds, sometimes with astonishing success, or confused failure. In his *The Sound of Light in Trees*, he simply plays back a slowed-down recording of bark beetles communicating at ultrasonic frequencies. These pieces show how emergent structure exists in nature, and suggest something profound about the relationship between

natural behavior and music.

Agostino Di Scipio is a composer whose work is like Dunn's in that it is inspired by ecology, although he does not interact directly with ecosystems. He has written extensively about dynamical systems and iterated functions [45]. His music is often made with custom programs that synthesize audio on the sample level. The use of noise and feedback in Di Scipio's music has gesture and structure because he writes algorithms at the sample level that are designed to have perceptible effects at larger scales of time. This enables the algorithm's emergent behavior to be heard. Di Scipio has discussed the influence of Xenakis and Brün on his work. He has also written of his views on their music [45]. For Di Scipio, the importance of a computer system (program) is found in its emergent behavior, *and* in the analog it has to natural systems. Di Scipio's carefully crafted music contrasts vividly with the nihilistic noise music made by Merzbow or Incapacitants, who are more directly interested in the philosophy of meaninglessness within sound.

2.2.2 Data Bending, Glitch Music, and Sonification of Data

The creation of sound to communicate or provide insight into a piece of information is called sonification. There is a related practice that has come to be called 'data bending,' where some data, such as an image, is sonified without the intent of transferring any of the meaning the original data represented. Data bending is closely related to glitch music, which came to

be in the 90s. Some examples are Oval, Yasunao Tone, Ryoji Ikeda, Farmersmanual, Florian Hecker, and Kim Cascone, who is also one of the few authors to have written about glitch and data bending in an academic context [46].

This type of music is not only created by the 'misuse' of data, but it is also about the misuse of data. However, both glitch and data-bending are not strictly noise; although the meaning and intent of the data has changed, one can often hear some kind of structure in the music. If the structure exists in the original data, then it has to be said there is a meaningful relationship between the original data and resultant sound. For example, using a tool like *Soundhack [47]* to sonify images, it is difficult to 'hear' the original image, but one can hear an identifiable difference between JPEGs and GIFs. In some sense, one might say that data bending techniques exemplify the resilience of data to retain structure and pattern while removing content.

Other composers of algorithmic music have taken approaches that do not aim for the 'misappropriation of data,' but to provide insight into the data in its sonification. One example of this is Charles Dodge's *Earth's Magnetic Field*, which took data from magnetic observation stations gathered over a year, sets up a sonification scheme that 'played thru' the data in fourteen minutes. Being a sonification, the sounds themselves involve tones that come from the western tonal scale and comb filters. These choices, along with the

sonification scheme, are compositional choices. They have little to do with the actual physical phenomenon of Earth's magnetic field or the data it produced. However, since Dodge made these choices carefully considering the data, it can be clearly differentiated from data bending. One effect the piece has is the transmission of a concept that is referenced by, but not present in the original data; the scale of size and time of the earth and a year are presented in a room over an amount of time humans can appreciate music in.

2.2.3 Live Coding

Some computer musicians present their music in the form of laptop improvisation. One criticism to this approach is the claim that the keyboard/ screen combination, if used as the main interface device that the performer interacts with, presents an unexpressive performance that leaves the audience wondering about the relationships are between the performer, the computer, and the sound in the room [48]. Live coding [49], or, on-the-fly-programming addresses this criticism with the performers actively writing code to control the music (or video) during the performance. The code itself is often projected on the screen for the audience's appreciation. Live coding groups tend to have names that pay tribute to geek culture: examples include klipp_av, who use *SuperCollider*, aa_cell, who use *Impromptu* [50]. *ChucK* is used for on-the-fly-programming by the Princeton Laptop Orchestra (PLOrk) [10], [51].

The desire to share the code with the audience could have several motivations for a composer. Some of this desire may be traced back to the open-source software community, which serves the computer music community with tools such as *SuperCollider* and *Audacity*. Another possibility may be related to ideas from process music. The developmental process, as well as the process of 'getting it right,' may be something the composer wishes to convey to the audience as a complement to the audio and video. Lastly, it is important to note that one of the cultural values of live coding is the idea that programming itself, through the understanding of data structures and algorithms, can be an expressive means of making music.

2.2.4 Data Structures and Algorithms for Music

In his music, Charles Ames explored low-level programming ideas, including data structures and recursion. From his description of his piece entitled *Concurrence [52],* one gets the sense that he is describing data structures as not only a means to implement an end, but also as providing the means to extend the aesthetic end:

> The set of composing programs developed for the U.S. pavilion at Expo '85 in Tsukuba, Japan, and the programs for my 1986 solo violin piece, Concurrence, employ 'linked' data structures, such as 'lists', 'trees' and 'networks', to represent the complex interrelationships between elements both of the musical scores themselves and of the musical 'knowledge base' consulted by the programs as they fabricate these scores.

Ames also was interested in statistical distributions, laying out a practical survey of distributions in an article [53].

John Chowning's *Stria* is well-known example that incorporates the mathematical properties of the golden mean, using recursion as well as his research in frequency modulation (FM). Chowning was the first to understand how to manipulate the modulation frequency and index in FM to create predictable partials. FM can create harmonic and inharmonic partials by creating sidebands. If the modulation index is high enough, the sidebands can also create their own sidebands, creating a self-similar structure. His use of the golden mean allowed these self-similar partials to align with partials of other notes placed along 9 tones in a skewed octave ratio of 1:1.618, instead of the usual octave ratio of 1:2 [54]. This references the way natural harmonics align when played at consonant intervals in the western equal temperament. Chowning also employed recursion to create self-similarity, density, and a means of transition between the individual synthesized elements. The entire piece is rigorously formalized as a program in Stanford Artificial Intelligence Language, controlling elements from the low-level parameters to the larger time scale of the sections.

The Hierarchical Music Specification Language (HMSL) [17] was successful in bringing several low-level programming concepts into the foreground of the computer music world, including data structure hierarchy and polymorphism. Composers using HMSL wrote a wide variety of pieces from self-organizing pieces (Larry Polansky) to networked computer music based upon fusion

reactions (Phil Burk). Nick Didkovsky [55] also wrote pieces that took advantage of data structures. HMSL, and the subsequent Java Music Specification Language (JMSL) claimed to be designed to be non-stylistic, which means that the music written with it did not have any supposed social push to be about data structures or programming, or anything else for that matter. That being said, the low-level access to audio and programming that it provided was at least partially responsible for attracting programmers and some hacker culture (Didkovsky has a piece entitled *Fast Fourier Fugue*.)

Within this music programmer culture lies another interesting group worth mentioning. The Hub [56] was six musicians that performed improvised music in the 80's and 90's via a network. Each of the six members wrote their own software, but their programs could communicate with shared memory, (and later, MIDI messages.) The emergent structure of the network being the main structural element (as opposed to the individual programs) is a characteristic of The Hub's music that provides insight into the way networks–human, abstract, and cybernetic–have a structure that can be understood through music.

2.3 Non-Musical Work

2.3.1 Architecture and Design

The importance of structure is obvious in architecture. Christopher Alexander was one architect who explicitly stated that there was far more to

a structure than its physical components. In his *The Timeless Way of Building [57]* and *A Pattern Language [15],* he discusses the philosophy of patterns and structure. *A Pattern Language* describes a complete generative pattern, with a few pages detailing each of its 253 patterns. Some examples are high-level patterns, such as 'country towns' or 'eccentric nucleus,' which encompass more abstract patterns such as 'network of learning' or 'night life.' They can also contain low-level patterns such as 'open shelves.'

Alexander goes over the aesthetics of making a good design with these patterns. Many of these concepts apply directly to computer science and programming. For example, the reuse of and ability to combine patterns is something that is useful in programming. In *A Pattern Language,* Alexander stresses his idea of 'compression.' That is, by using a single material or pattern in such a way that can be used with and by other patterns. He believes this to provide structure and 'life' to the design.

The Timeless Way of Building invokes elements of aesthetics. When discussing structure and design, Alexander uses poetic terms, such as 'the quality without a name.' He discusses buildings as being 'alive' or 'dead.' Throughout the text, Alexander refers to natural systems, from the development of an embryo to the differences between atoms. One gets the sense that he uses these systems from nature, physics, and ecology not only to justify his methods, but also to suggest there is something natural and

transcendental about the systems behind the phenomenon.

Alexander's earlier *Notes on the Synthesis of Form* [27], provides a foundation for the pattern language ideas. Form is discussed as an emergent property of fitting an idea into constraints. Constraint solving is discussed in all three of these texts, but it is most technical in this one. Alexander uses the solving of constraints to suggest that complexity might be best approached by an abstracted design.

Alexander's ideas have influenced architects, computer scientists, and designers. The software known as *Processing* is the result of media artist, computer scientist, and designer, John Maeda. Maeda's book, *Design By Numbers* [16], looks at using algorithms to control visual design. Maeda's 2004 book, *Creative Code,* presents the work of many media artists that work with code, interspersed with his thoughts about the field of media art, such as the following:

> A strange reverse phenomenon is in motion today: As programming becomes easier and more accessible, the tools for expression are becoming more complex and difficult to use. Programming tools are increasingly oriented toward fill-in-the-blank approaches to the construction of code, making it easy to create programs but resulting in software with less originality and fewer differentiating features. The experience of using the latest software, meanwhile, has made even expert users less likely to dispose of their manuals, as the operation of the tools is no longer self-evident. Can we, therefore, envision a future where software tools are coded less creatively? Furthermore, will it someday be the case that tools are so complex that they become an obstacle to free-flowing creativity?[58]

Maeda frequently refers to the relationship between programming and art, taking note of issues that involve complexity. This is related to the levels of

access a programming language allows a programmer to use. The previous quote illustrates the designer's perspective on appreciating programming languages as tools for creativity.

2.3.2 Program Auralization

Programmers spend half of their time writing code, and the other half running the code to debug it and find out why what they wrote was wrong. As a result, many debugging techniques have been developed. The breakpoint-based debugger connects code to the program while it is running. On the other hand, simple 'printf' console style debugging is useful for some situations. These techniques are generally about giving the programmer visual feedback about a program's state. One relevant anecdotal programming technique evolved the late 70's and early 80's, when programmers could actually hear structure in certain patterns from the processor when it executed code such as nested loops. This is another type of feedback that programmers found to be useful debug programs.

The visual text layout of code is largely spatial, and program execution is largely temporal, as in sound playback. In 1982 Sara Bly published a paper entitled *Presenting Information in Sound [59]*. While it referenced the work of many important computer music researchers and composers, (including Tenney, Rissett, Mathews, and Chowning,) the primary purpose of the research was to develop non-musical applications for sonification. Bly's work

sparked the interest in a field now called program auralization [60], [20], [61]. The idea behind program auralization is to use sound as an aid in understanding program code and flow. There were other motivations as well, such as the difficulty of debugging asynchronous events in parallel programming. Most of these auralization systems were deliberately non-musical. They are surveyed in a paper by Vickers and Alty [62], [63].

Users of program auralization systems complained that even though the sounds produced by the auralization were capable of transmitting information, they were annoying and tiresome. The claim is not superficial, since presumably an annoyed, tired programmer will accomplish less than a less-annoyed, less-tired one. This led Vickers and Alty to experiment with systems that used characteristics of tonal music to provide the users with patterns that took into account the previous cultural experience in a context such as that of the western tonal system.

Program auralization has not become a big component of the average programmer's debugging kit. For example, it is not a part of popular integrated development environments (IDEs) such as MSVC, XCode (which includes gdb,) or Eclipse. Still, the convergence on musical and cultural contexts for sonification is significant and relevant to a discussion about writing programs that create music. The main difference between using data structures for composition and Vickers and Alty's work is illustrated in their

claim that the sounds that come out of their auralization are music without intent:

> The intentional product of an auralisation system is the communication of information or knowledge with music as the carrier. The music itself, inasmuch as it exists as an entity in its own right, is not the intentional product but a by-product of the auralisation process. Therefore, whatever music systems and aesthetics are employed they must not detract from the prime purpose which is to communicate information.

Vickers and Alty do not discuss musical structure and hierarchy in detail. They imply that this kind of work can help debug data structures and algorithms that have arbitrary complexity. The relevant assumption here is that data structures are understandable through sound and music.

2.3.3 Aesthetic Computing and Software Art

The aesthetic computing movement is related to program auralization as well as design (John Maeda led a group called "aesthetics + computation".) But one of the main differences is that its applications cover a broader area. The aesthetic computing movement contains work very similar to program auralization, such as designing visualizations for data-structures. It also encompasses discussions reminiscent of cybernetics, using and challenging ideas from human-computer interaction, as well as interdisciplinary art about stem-cell modeling. The aesthetic computing manifesto was published in Leonardo in 2003. Here is an excerpt from the one-page document:

> Computer programs and mathematical structures have traditionally been presented in conventional text-based notation even though, recently, substantial progress has been made in areas such as software and

> information visualization to enable formal structures to be comprehended and experienced by larger and more diverse populations. And yet, even in these visualization approaches, there is a tendency toward the mass-media approach of standardized design, rather than a move toward a more cultural, personal and customized set of aesthetics. The benefits of these latter qualities are: (1) an emphasis on creativity and innovative exploration of media for software and mathematical structures, (2) leveraging personalization and customization of computing structures at the group and individual levels and (3) enlarging the set of people who can use and understand computing.[14]

Media and digital art are well known 'genres' of art, with competitions and festivals such as CyberArts, Rhizome, and many others. Florian Cramer has written extensively about software art. He calls for a questioning of stereotypes that the audience carries into the experience of digital and media art. He takes the stance that media art's tendency to hide the process and the code with the GUI might be restrictive. The following excerpt is from his paper *Software Art*:

> Software art means a shift of the artist's view from displays to the creation of systems and processes themselves; this is not covered by the concept of "media." "Multimedia", as an umbrella term for formatting and displaying data, doesn't imply by definition that the data is digital and that the formatting is algorithmic.[13]

This post-GUI tendency to place value upon programming and process has elements in common to other related musical work already discussed, from process music to live coding, as well as to the author's pieces presented in chapter 4.

3 DATA STRUCTURES AND ALGORITHMS AS COMPOSITIONAL TOOLS

The main focus of this chapter will be on extending programming concepts and data structures to act as inspirations for musical structure. The advantages of applying programming and data structures to music will be considered in the process of designing musical structure, as opposed to solely in the implementation of it. Many pieces of computer music have been written using data structures, as shown in the previous chapter. In the majority of these pieces, the use of data structures is necessary for a popular framework, be it genetic algorithms, Markov models, or flocking [42]. On the other hand, there are other earlier examples such as Xenakis's *S-10* [36], which use data structures to implement a certain music-structural design. Both cases are important and will be considered in this chapter.

3.1 Programming and Music

Programming offers an array of tools for tackling problems of many sorts. It is interesting to note that many people decide to study programming, even though the knowledge of programming does not directly accomplish anything – its abstract concepts must be applied to a problem, which probably has

little to do with programming. Or is this the case?

There are certain fundamental elements of programming that have much in common with music. These shared elements are hierarchy, reference, and synchronicity. This section will be an attempt at discussing the various ways that low-level programming constructs might be applied to give a piece structure.

When applying numbers, (and thus data structures,) to music, this section's title is one of the questions frequently asked of the algorithmic composer. The question is about how the data has been interpreted to be musically useful. So far the discussion of using data structures and programming as compositional tools has been abstract. Now, an explanation of why certain concepts in programming and data structures have interesting similarities to certain valued musical concepts will be presented. These relationships describe a mapping between music and programming that will be useful to keep in mind throughout the remainder of this chapter.

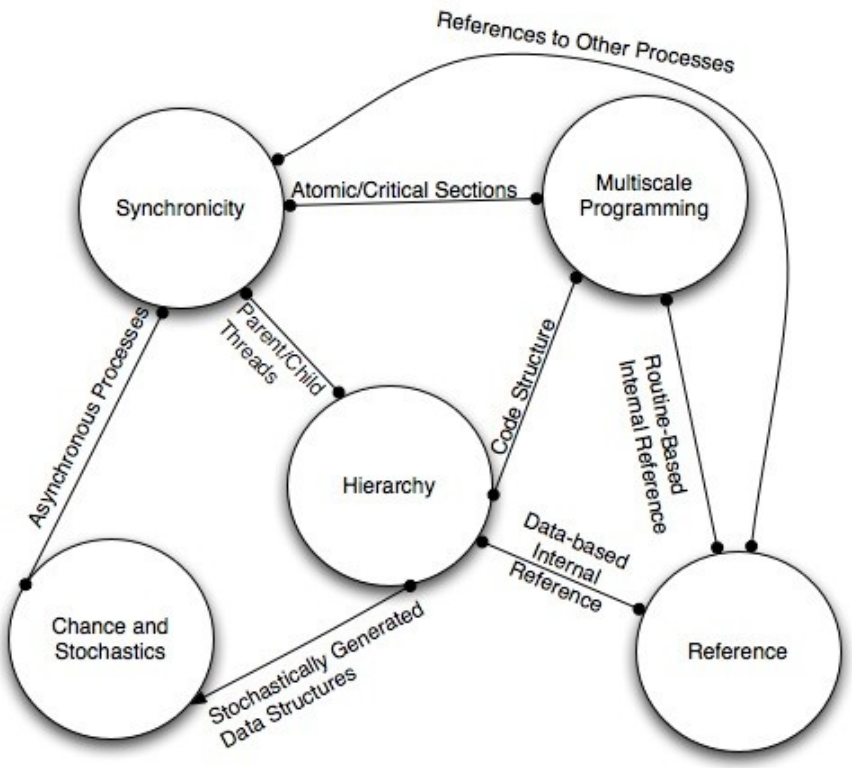

Figure 2. Software concepts presented in this section and their relationship to each other. The main concepts are presented in circles, and their commonalities are presented as connecting lines.

3.1.1 Hierarchy

Music generally can be analyzed as having hierarchical components. Traditionally, movements are broken into sections, which are broken into phrases, which are broken into individual notes. There is a long history of analyzing music as embodying hierarchies. Most analysis of musical structures have a tree or graph like representation for hierarchy. Tenney and Polansky's work on analyzing hierarchical parameters using explicit terms

such as 'hierarchical levels,' is a good example of this. Another is Schenkerian analysis. Creating musical hierarchy through data structures has been explored by Xenakis, Chowning, Tenney, Ames, and many others. It is so large a topic that it needs to be explored further.

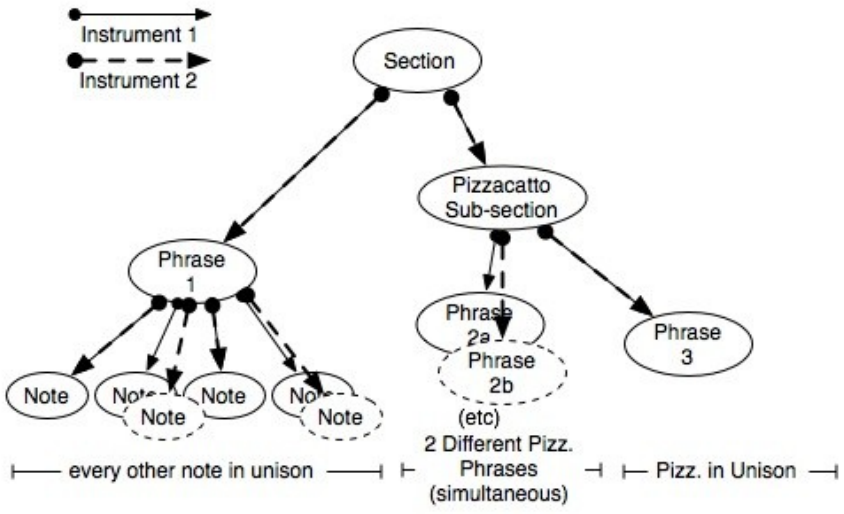

Figure 3. Hierarchical music analysis (oversimplified for example.) There are two types of hierarchy represented here – one for duration, and one for instrument. Voices (instruments) sometimes share nodes to signify a similarity between the two. For example, both instruments have their first phrase at the same time. However, they differ on some notes (notated by the separation of dashed and solid lines.)

Data structures define a hierarchy of inclusion – For example, a tree node contains other tree nodes that are its children. All of the basic programming data structures, the tree, the linked-list, the stack, etc., can be drawn on a two-dimensional sheet of paper. Computer networks have lattice-like data structures, and are generally taught as two dimensions. This is because in this case they represent a two dimensional space (i.e., on a map of the Earth.) In graphics programming the mesh, which is essentially a network of

verticies, often represents three dimensions. All of these data structures, whether they model a physical body or hold stock market data, have hierarchies that are dimensional.

In writing music that focuses on hierarchical data structures, I became interested in data-structures that were difficult to diagram and sketch on a 2-D surface. It may be relevant to refer again to architecture, but this time, instead of theory, practice. In designing structures that exist in a three-dimensional world, architects are faced with the problem of perspective. To describe a three-dimensional object on a plane, it must drawn either from several angles, or with an actual three-dimensional scaled-down model. Now consider the case of hierarchy in music. With music having a multiplicity of parameters it seems that there might be many simultaneous hierarchies, and that these hierarchies may have connections to each other at various points. This makes a good case for using multiple dimensions within the data structure hierarchy. *Tree Music,* in section 4.1., is a piece by the author which links several trees together to form a multidimensional hierarchy.

3.1.2 Reference

Reference is a large part of music and programming, yet most computer music does not take advantage of programming references as musical devices. Compositional programming paradigms for music have so much abstraction

that it is difficult to do so. Pieces are often implementations of instrument/score models, unit generators, or some other cognitive model. There, the idea of a using a low-level programming pointer as a referent or to define a relationship becomes unintuitive.

There are many types of references in music. They often involve the revisiting of previously encountered segments of music. The ideas of restatement, development, and repetition can be viewed as the application of a process to a reference. These types of reference are internal to the music. The self-referential constructs are almost purely referential, since it is about references that have references to others. The idea of quoting or borrowing from an external piece of music, (often by another composer,) is also a very referential idea. External references can be non-musical ideas as well. For example, there is an explicit reference to a horse galloping as well as to the idea of the feeling of 'tension' in Schubert's *Erlkönig*. Perhaps the most explicit example is the referential capacities of words, which often have nothing to do with sound or music. In addition to song, this can be done with references to literature, as Berio did in *Sinfonia* and *Ommagio a Joyce* to reference Beckett and Joyce.

In the experience of music, the listener makes a huge number of references to all the music and life experience he had already experienced. External references are always implicitly there at both the macro and micro scale. For

example, any piano piece references the 18th and 19th centuries, as well as the idea of something being struck, with each note, whether the composer or listener is aware of it or not. The composer can focus on using external references as compositional material. The interested reader might refer to Tenney's similar ideas of the 'subjective and objective set' as defined in *Meta+Hodos[64]*. Tenney's ideas are about expectation and analysis, but there is still much common ground.

C programmers will be familiar with the idea of reference via pointers. Pointers allow for arbitrary levels of indirection in their references, (meaning it is easy to construct a reference to a reference, and to it another reference, and so forth.) The idea of 'levels of indirection' is complicated, because it seems to mix the idea of reference with hierarchy. Programmers found this idea to be powerful, but difficult to master. This is well illustrated in the canonized quote from David Wheeler, "All problems in computer science can be solved by another level of indirection," eventually extended to "...except for the problem of too many layers of indirection" [21].

Explicitly and implicitly, reference is something composers have traditionally used in their music, but usually not by using C pointers. If an entire piece of music is programmed, the composer using data structures to represent musical structure can simply attach a pointer to certain parts of the music to be recalled or modified at a later time. Of course, musical

reference is more complicated than that because of its semantic value. Besides a pointer to the referent, the programmer will need to implement an action or relationship that defines what the reference means.

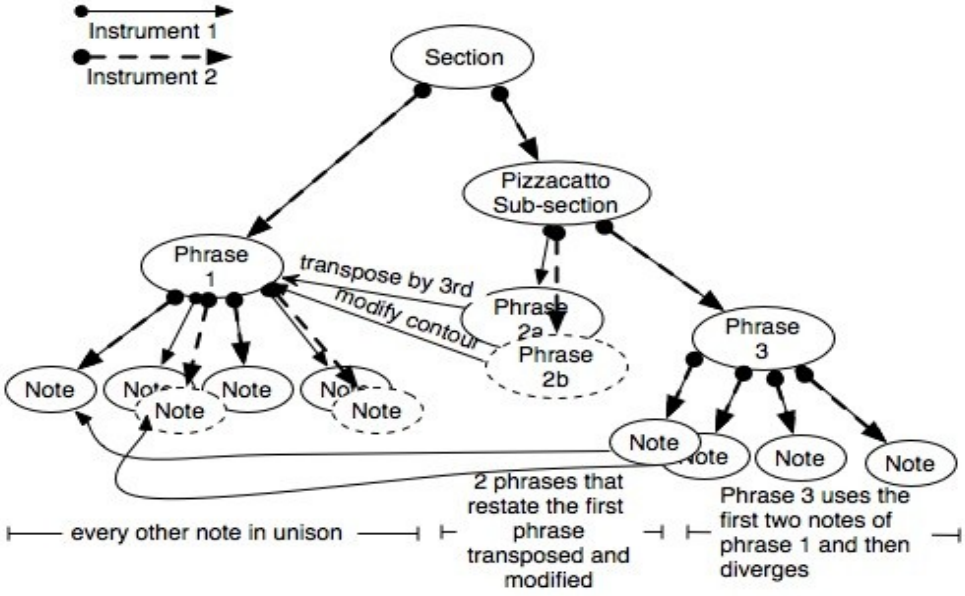

Figure 4. Musical structure with references. The figure is similar to the previous figure on hierarchy, but differs in that here there are references that break hierarchy. Note that the references also define a semantic meaning (e.g. phrase 2a is phrase 1, instrument 1, transposed by a third.)

Implementing internal references is a simpler task than external references. This is because the internal referent is purely musical or structural. With external references, the referent could be anything, from musical, to physical, to cultural, to philosophical. When the reference is external, there is the problem of how to create the representation for the referent. For this reason, the easiest starting point would be to explore internal references using programming, and external references the way composers have in the past, intuitively. However, with the increasing availability of modular programming libraries and web apps, it is becoming more conceivable to

write a system that manages external references in a meaningful way.

Note that we have not departed far from the previous section. If references create a link, they create structure, and structure defines a hierarchy. *Tree Music* in section 4.1 takes advantage of this relationship, and references the hierarchy in tree structures. *Dict* in section 4.3 explores external references by using a dictionary.

3.1.3 Synchronicity

Multithreaded programming offers not only the ability to run processes simultaneously, but also provides the programmer, (via the mutually-exclusive lock, or mutex,) a control over synchronicity. It does this by dictating which parts of the processes will wait for or join up with other processes and which parts will run independently. In contrapuntal music one often finds musical lines that produce simultaneous events that split up and come back together. This occurs at various levels of the time-based hierarchy. Multithreaded programming exists for the purpose of having multiple processes run with synchronous behavior. This is analogous to the synchronicity in music that creates counterpoint and polyphony, as well as the way independent entities operate within society.

One can also discuss synchronicity in terms of mutual exclusivity between asynchronous processes, i.e., events that cannot happen at the same time, or a subspace that cannot be occupied by more than one thing at a given time.

The mutex provides this option in programming. Analogous behavior can be found in well-formulated systems of musical counterpoint. For example, Fux style counterpoint has many rules, such as the 'no parallel fifths' rule, which place constraints on one line based upon another. This kind of system can be implemented with a backtracking programming technique, but threads could also be used here. In the case of threads, a process that generates the pitches for a voice could place mutexes upon the notes that its current pitch choice causes other voices (processes) to exclude from their selection.

Note that we still have not departed far from the previous sections. Dependencies between otherwise independent processes are defined by referencing other processes, and the hierarchy – which process waits, or creates other processes – create a structure that can become multidimensional.

3.1.4 Stochastics and Random Behavior

The ability to generate masses of pseudorandom numbers is one of the prominent features of using a computer for music. Using stochastic distributions [53], such as those of Xenakis [35], it is possible to exert compositional control over a collection of random numbers.

Another way to use random numbers is in the logic of the program. Many (non-video) games implement this process via dice, to add some unexpected, but constrained behavior. Role-playing games, such as *Dungeons and*

Dragons, or their online counterparts, multi-user dungeons (MUDs), construct elaborate systems around the dice rolls, such that they can model a series of life-like events from the blows and damage thrown in a fight, as well as whether or not one character will find another character interesting.

The use of randomized logic is not limited to video games. One example where this has emerged is in computer networking protocol. In a network, many clients need to use the same cable, (for example the cable coming out of a hub that five computers are hooked up to.) Only one can use it at a time, so communications must follow a mutually exclusive protocol. However, it is not possible to use a mutex object, since this would need to be accessible by all computers on the network at all times, which is impossible if only one computer can access it at a time. Programmers found that the optimal solution, (that produced the shortest amount of waiting) was to have the network clients check to see if the network was busy, then wait a random amount of time between 0 and 125 milliseconds. A fixed amount of time would cause the clients to get into unproductive rhythms with other clients. A system to manage these rhythms could also be designed, but it might not scale as more clients are added. It is clear that a random wait is the most elegant solution to this problem. This means that there may be some problems in musical structure that a randomized logic could benefit.

The die based approach to random numbers can and has been applied to

music. Although we can trace the actual use of dice in music back to Mozart, John Cage's experiments with chance are probably more appropriate here due to their number and documentation. Cage used many objects that were not dice, from the *I Ching* to transparencies that could be dropped upon other transparencies to determine musical parameters such as loudness and event times. Cage also made the connection between chance and the indeterminate components the performer introduces into the piece. The relevance of this connection is that it makes an argument against the viewpoint of random numbers on a computer as something alien to all music that does not use them.

There is a science to understanding stochastic processes [65], but there is also an art to finding the right values and structures for a particular application of stochastics. It becomes more important if the code uses simulated dice rolling to control explosive code flow techniques such as branching recursion, as in the generation of a tree, because things can blow up or die out faster than estimated. If this happens, the programmer can go back and tune the dice roll (by changing the number of sides, or adding modifiers,) or, simply put a cap or minimum in the program. However, it should be noted that the latter type of solution is truncating the ends of the dice. This is totally acceptable, but a smoother solution might be, for example, if the code branches out too quickly, to reduce the probability of

branching as the depth increases.

3.2 Multiscale Programming

The preceding section was about general programming concepts and their relationship to music. This section is about how these concepts are structured, and in what scale they are applied. One objection to score models of music is that this kind of notation makes it more difficult to get 'inside' of the event to manipulate it. On a traditional score, a quarter note specifies a high-level event. The performer controls the lower-level parameters such as the actual volume, duration, and brightness, even if it is not notated. In an entirely synthesized music generating program, controls such as envelopes will also be important to control the gesture for the entire duration of the note. In this regard, the program is always effecting the synthesis output – this raises questions of where the event really is. Another way to look at it would be to say every sample is an event in itself. However, unless the intended result is noise, each of these samples must have a relationship to the preceding ones. This essentially means that for music written with programming languages, the use of structured programming is required at every level of scale that the composer deems important.

Another advantage of synthesis closer to the sample level is that it is possible to build structures that have the kind of references discussed in

3.1.2. to recycle or define some kind of relationship to a musical parameter at a previous point in time. This does not exclude synthesis techniques such as soundfile sampling, but it will require some kind of analysis/parsing of the soundfile to create a meaningful structure to be used in this fashion.

Work in granular synthesis is also concerned with sound at or near the sample level. In *Microsound [66],* Roads presents the concept of a "multiscale approach to composition," which means to "insert, delete , rearrange, or mold sounds on any time scale." The majority of Roads's text concerns granular synthesis. Granular synthesis is related to multiscale in that it controls microscale audio at the macro level. This approach enables one to synthesize small packets of sound to specify arbitrary amounts of grains of sounds using stochastic means. However, this does not take into account *how* to control and modify the granular synthesis parameters at larger levels of scale. This leaves the composer to decide what to do with the levels of scale from slightly above the microscale to the large-scale form of the piece.

If the composer wants to be able to have fine control over all the levels of time scale, he will need a program to manage the parameters of the synthesis algorithm over the duration of the piece. If cascading levels of time down to the synthesis algorithm are used in this fashion, the instrument/score (or unit generator/synth message) division will be much harder to discern. This approach may be useful for this very reason. It could be called *multiscale*

programming for its similarity to the way Roads's multiscale approach to composition respects the cascading levels of scale in sound. There are advantages to simultaneous high, low, and mid-level programming; up till this point mostly low-level programming concepts as tools for constructing higher level musical constructs have been presented, but that is useful only if the low-level programming is structured with elegance. That requires higher level programming.

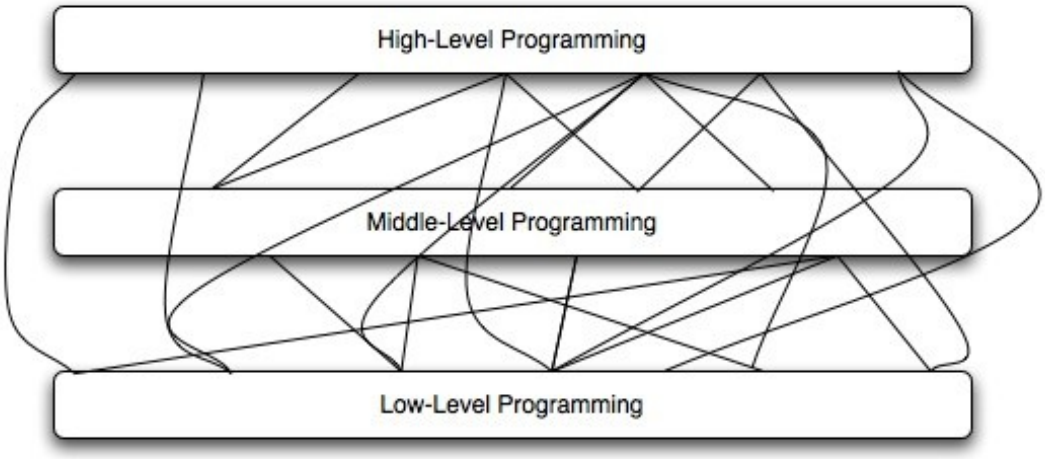

Figure 5. Structure also exists in the high-level to low level programming. If considering using programming structure as being useful for creating musical structure, it is worth noting that focusing on just the high-level or low-level programming will essentially block this opportunity by 'cutting the connecting strings' between the levels. Likewise, closed source, or 'black box' use of tools hide the inner workings of the code, and thus, the ability for the program to create structure by attaching references to the structure within the tool's code.

The major music creation programming languages currently in use have a design that makes multiscale programming nearly impossible. For example, *SuperCollider* allows access to elements of low-level programming, but there are three modes of 'coding' (SC language objects, synthdefs, and C++ unit generators,) which separate the levels and do not allow the linking between

them necessary for multiscale programming. In non-musical applications important program structure *sometimes* exists in the relationship from high to low level functions and data structures. However, if one's interests are in using software structure to manipulate musical structure, it seems the programmer will need convenient access to all of these levels, firstly, to understand the complete structure so that he can correct and modify it, and secondly to be able to have convenient control over any level of the sound and program.

3.3 Analysis and Aesthetics

All of the concepts mentioned have one thing in common; they ultimately affect structure in music and program flow. It would be beneficial to have a measure of structural and formal complexity to evaluate the music and the procedures used to generate it. It is important to recognize that ultimately the evaluation of music is to a large degree subjective, and based upon cultural context and previous experience. Therefore, it does not make sense to formulate a general definition for musical evaluation, since both the listener and composer will bring different criteria to the evaluation. Rather, it may be more practical to define a handful of characteristics, which are present in the music/code relationship, and which may be musically interesting in one's subjective experience of music and programming.

3.3.1 Minimum Description Length and Kolmogorov Randomness

Consider the following sequence of numbers:

2 3 5 9 17 33 65 129 257 513 1025 2049 4097 8193 16385 32769 …

Note that the string has a pattern that can also be described as:

$f(n) = (f(n-1) -1) * 2 + 1, f(0) = 2$

which is more compact, requiring fewer keystrokes than typing all the numbers in the original sequence. There is an infinite number of alternate descriptions for this sequence. The minimum description length (MDL) is the character count of a description which is the smallest possible in terms of a given description language, for an input string.

Consider the following string:

1 3 3298 83749 532 1837900 12754 48123 28 9393 6923 …

There is no obvious pattern in these numbers, so the smallest description that intuition provides is simply to write out the original string. However, one could formulate a representation that can recreate, not the exact values of the original string, but rather, certain characteristics of the sequence. The string might then be described in loose terms of characterization, such as "random numbers under 8 digits," and perhaps further constrained with additional characteristics such as "the difference between adjacent characters usually alternates between positive and negative values." This is roughly how audio analysis/resynthesis works, from sinusoidal modeling to MPEG encodings, since the resynthesized audio differs from the input. The benefits

of these systems are twofold. One, the representation can be reduced in size, and two, the information is reinterpreted to be something *perceptually* significant to humans.

Given a string of notes, or sounds, one might be able to guess at the process by which they were generated, and in some cases, one might actually hear the process directly in the music, (as in process music.) Processes are very similar to the 'characteristics' discussed in the previous MDL discussion. Process can be made more obvious with music composed via programming languages. The composer and listener may then compare the experience of the music to the actual process used to generate it.

3.3.2 Refactorization and Computational Complexity

An exercise the composer can use to improve his code is to see if the data structures and algorithms used to create music can be reduced to create a perceptually identical result. This is known as refactorization. Refactoring can be used to discover discrepancies between the programmer's intent when writing the code and the result. The presence of structure or hierarchy in a sonified data structure does not guarantee the music will exhibit them. If a refactoring produces not only a perceptually identical, but exactly (physically) identical result, then the algorithm is needlessly complex. This might point to some structural assumptions that need to be revisited.

The field of computational complexity [67] does not directly concern itself

with how to refactor code and reduce complexity, but it is related in that it provides quantifiable orders of complexity. By analyzing an algorithm and its program flow one can determine a "class of complexity" in which the algorithm runs with respect to the input size. When a reduction in this class (called 'big-Oh' notation,) occurs in a major problem, (e.g. Cooley and Tukey's work on the FFT [68],) computer scientists get very interested, and not only because the algorithm runs faster. A reduction from a more complex class to one less complex does not guarantee a speed increase for all inputs, (although it does for very large inputs.) The reduction in complexity indicates that a new insight has been gained about the *structure of the problem*, which the old solution's implementation failed to grasp.

Work in computational complexity, MDL, and Kolmogorov randomness [69] suggests that problems have objective complexity, and therefore an objective structure for an ideal solution. This statement might interest experimental music composers who claim that their work attempts to solve a musical problem. If the piece is written in code, the composer should be aware that his implementation of a problem, (his piece,) will have a structure that bears a relationship to an objective structure. An objection might be made that the aesthetics of failure, or Dadaist aesthetics can override the importance of paying attention to this objective structure. Another perspective may be to apply Another Level of Indirection to say in taking that stance the critic is

actually defining the problem in terms of the objective structure from a rebellious or Dadaist viewpoint.

Tree Music (section 4.1.) went through several phases of refactorization in debugging. Debug tools check assumptions and produce refactorized code. This cleaned up the structure and made it more intuitive to extend. *Limited Resources* (Section 4.2.) uses something related to refactorization in the way it draws objects. Traditionally the animation update of interactive objects takes place in a central location. Because it is computationally intensive, when the number of objects get to a certain density, the screen starts to lag. The objects to be animated run their updates on their own threads. When these threads were blocked, or the computer became slow, it only affected some of the objects, since some threads were able to complete their simple individual task.

3.3.3 Overfitting

The fields of artificial intelligence and statistics are concerned with analyses that "overfit" their objects [70]. If one is too ambitious, or too strict with one's criteria for finding or guessing the process used to generate some section of a piece music, one will come up with a large number of rules that may describe the section very well, but will often fail to describe other sections of the same piece, even if the other sections have much in common with the section that fits the analysis. The basic concept of overfitting, then,

is the idea that treating a coincidence as a rule will defeat the generality of a model. Since musical analysis is a type of modeling, overfitting applies to the analysis of musical processes, especially those that are rule-based. Xenakis's main claim against integral serialism was that its composers were overfitting the pitches. For Xenakis the complexity produced a musical entity where the individual pitches (points) were not that important to the musical effect.

But how does one decide when a series of notes is so complex that the individual pitches becomes unimportant? The idea of Kolmogorov randomness may serve to illustrate the extreme case. Kolmogorov randomness is, interestingly, not defined in terms of indeterminacy. A string is said to be Kolmogorov random if the number of characters in the shortest program that can be written to generate the string is at least as long as the string itself. The sequence of pitch in a serial piece may not be Kolmogorov random, since the programmer could use rows and transformations to compress the program's length. But as a mind exercise, one can consider the length of the shortest program that can generate the pitch series, and take the ratio of it and the length of the pitch sequence to get a crude measure of a fuzzy version of Kolmogorov randomness. My belief is that one might find that the program flow, (or, structure of the code,) tends to be more elaborate and structurally interesting in strings that have shorter programs with smaller ratios.

While it is easy to imagine a 'random' sequence of pitches, it is a much more difficult task to imagine a 'random' structure or hierarchy. Structure and hierarchy define relationships between its parts, and so, in an analysis of music especially, they provide semantic value through reference. It would be awkward to say that semantical structure and randomness are opposites, but they do contrast, if not exclude each other. The structural semantics of a program define the interactions between the parts. They allow the programmer, (and the listener,) to break the problem, (and music,) into comprehensible chunks at arbitrary levels of detail. Since it is an error-prone programmer who defines the semantics for music, the comprehensibility of structure can fail when the semantics are meaningless to our perception. One example of this was explored in a piece written by the author entitled *Dict.* It took an English sentence and repeatedly replaced randomly selected words with a dictionary definition that did not break the grammar of the sentence. The resultant sentences were structural, (grammatically sound,) and complex, but so filled with useless information as to be incomprehensible (section 4.3.)

One of the fascinating things about minimum description length is that its lower bound is not computable. That is, for an arbitrary string encoding, there is no method that can verify that it is the shortest possible one. Turing proved that there are certain types of problems that are 'undecidable' [71] in

a way that resembled Gödel's work demonstrating the existence of unprovable truths [72]. If perceptual salience is used to further reduce the description length, the lower bound becomes even blurrier. There is an analogous problem in art music – the composer often wants the music to have a certain complexity to keep the listener interested, but if it is complex in the wrong way the intended structure will not be perceivable. Put in those terms, it is not so difficult, and perhaps comforting to see why minimum description length is not something an algorithm can decide.

3.3.4 Other Considerations

The previous section discussed methods for evaluating a piece to ensure that its complexity is not just in the process, but also the music. However, it is important to keep in mind that this evaluation must consider references of the physical work to its own process and culture. For example, there are no formal methods or algorithms for evaluating this aspect of art, yet artists do this themselves every day. The enormous scope of artistic evaluation makes it interesting and complex. A single evaluation of a piece of art is the experience of it *as it relates* to *all* the art and life experienced by the artist up thus far. These structural analysis exercises are not about the aesthetics of successful data transfer. They do encourage the idea of an implementation having structure that 'fits' the problem. But they do not give any metric for evaluating the problem itself. This is the part of composition that is strongly

related to experience, inspiration, and intuition. The composer will create a problem by constructing relationships between parts of his life and his concept of music. The piece is then called 'good' by the audience only if the musical problem was interesting, and its implementation apt, or simply because the composer is lucky, or simply played upon some cultural musical cliché that was fashionable at the time.

4 IMPLEMENTATION

4.1 Tree Music

In writing music, one is often faced with the problem of how and when to change a musical parameter. Attaching a seemingly random variable to a parameter can produce very predictable results, as is well documented in the Papoulis text on random processes [65]. Yet, there is some aspect of randomness and chance that composers such as Cage and Wolff felt could introduce new possibilities in music. They developed complex systems to control structure and form.

Tree Music is a piece composed in C++ using the author's audio libraries. The source is available at http://stria.dartmouth.edu/kdrepos/treemusic/ as an svn module. A binary tree has been modified to impose structure and form upon random interpolation of parameters. Each node in the tree represents a certain type of interpolation of one parameter from a start time to an end time, after which the parameter stays fixed at the end value for the node. The interpolation types can be continuous or discrete, exponential or linear, periodic or not. The children of the node combined take up the same time as their parent, and cause two different interpolations. The children individually have a random time duration, but their sum duration must

equal their parent's.

Children are generated recursively in pairs, or not at all. There is a certain percent chance that children will be generated, and this percentage is inversely proportional to the depth of the generating node. Children can interpolate any parameter, including that of their parent's or any other ancestor. Interpolation is done by coefficients that are concatenated. The amount that a node can interpolate a parameter is constrained by the minimum and maximum value specified at the outset of generation, and thus also by its parent's interpolation, for a child node cannot cause an interpolation that will cause a parent node to exceed the minimum or maximum. Once the range of possible interpolations is determined, a random value within this range is selected.

Additionally, when generating child nodes, an additional 'layer' can be generated if a random check is passed. This layer will generate an additional pair of children that will independently generate their own children from this point on. These layers represent a different tree that is identical up to this point in the hierarchy. This will cause a high-level similarity between the trees.

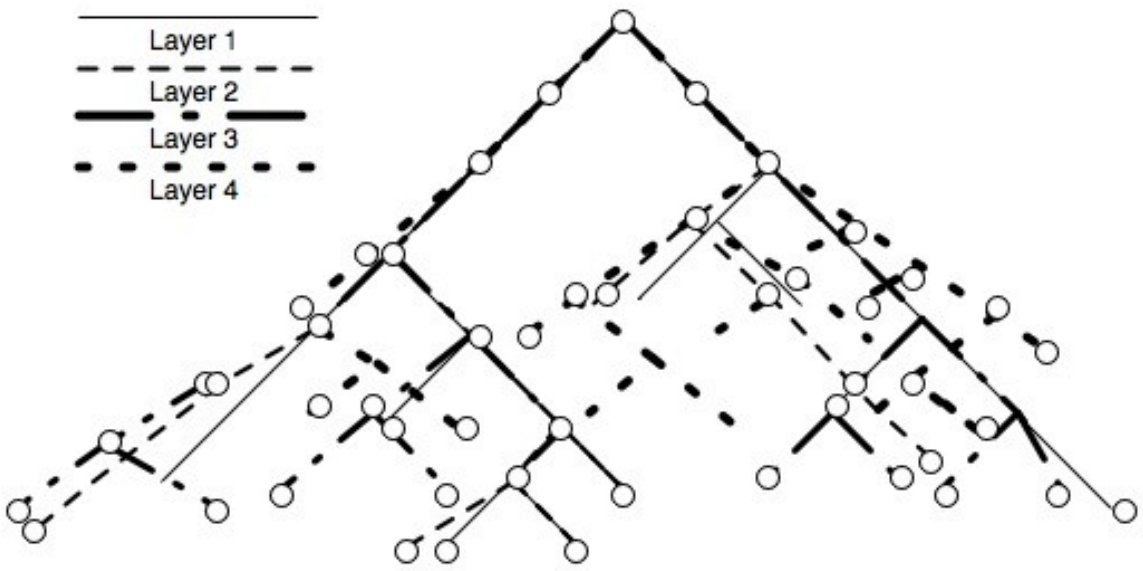

Figure 6. Simple *Tree Music* tree of four layers. Each layer represents the parameters for a noise-band unit generator. Each level of depth specifies a number of interpolations equal to the number of nodes at that depth for that tree. Note that layers can share nodes at the same depth, until a separation happens.

When it comes time to synthesize the trees, for each layer, all the instantaneous parameters are computed for each sample at each of the leaves. These parameters are fed to a synthesis engine to obtain the final waveform.

The apparent separation of structure and synthesis is somewhat misleading in that for very large depths in the trees, such as those with depths of 30 or more, there will be interpolations happening on the duration of a sample length. If the synthesis engine is responsive, this dense tree structure will have a direct effect on the sonic quality of the synthesis.

The first experiment was done using a very simple noise band synthesis engine that takes only three parameters – amplitude, frequency, and bandwidth. From four to eight layers were generated.

4.2 Limited Resources

A real-time audiovisual piece titled *Limited Resources* has been implemented in Objective-C and C++ as an experiment in exploring the possibilities of threads in music.

4.2.1 Multithreaded Programming

A simple definition of a thread is a process that runs and accomplishes a task independently, (for the most part,) of other threads. A mutex is a mutually exclusive lock on a resource, (be it a piece of code or variable,) allowing only one thread at a time to access it. Mutexes, therefore, are a form of communication between threads, telling them when to wait, and when to continue.

Threads are used in systems where there are limited resources, and mutexes when there are simultaneous requests for the same resource. They are also used when it is desirable to multitask. Some examples would be the banking system that controls deposits, web servers, and operating systems.

4.2.2 Natural and Musical Analogs

Some design patterns and data structures exhibit behaviours that have clear analogs to real life systems. This is even more important to consider with respect to artistic endeavours. These have great potential for referential use. In the case of multithreaded programming, there are numerous natural, physical, and musical systems that seems readily comparable to threads. A handful of these will be looked at here.

Traffic

In this human-imposed system, the drivers can be thought of as threads, and the intersections are resources which traffic lights control as mutexes. As traffic gets more congested, it is possible for cars to enter the intersection and not be able to exit. This can create a loop that causes gridlock – a behavior (deadlock) that also happens to threads in poorly designed systems.

Food

In ecosystems, animals typically have limited access to food. This access is limited, primarily by other animals, in their existence as food, and their capacity to consume food. Animals in the wild can be analyzed as exhibiting both thread-like and mutex-like behavior. The fact that they can act as both a resource and a consumer is interesting.

Ensemble Music

In many styles of chamber music, the players have parts with some degree of synchronicity and similarity. Some music also has constraints between the

parts that come and go, i.e. having contrapuntal behavior. Fux-style counterpoint is often taught literally using backtracking techniques, but one could use threads to accomplish this task in real time, with each thread being an individual part. Threads also have the ability to join up with other threads. This feature enables interesting hierarchical and synchronized behavior not limited to mutual exclusion.

4.2.3 Implementation

In *Limited Resources* threads are implemented as resource-seeking entities with finite lifetimes which are extendable by consuming resources. The seekers orbit the center of the screen, consuming whatever resources they touch by placing mutex locks upon them. Resources are represented as circles on the horizontal axis. Resources grow up to a maximum energy when they are not being consumed, and shrink as they are being consumed. They also have a 'recovery' time in which they provide no energy to seekers. As a seeker consumes a resource's energy it gradually moves towards the center of the resource. When subsequent seekers land on a resource as it is being consumed, they try to lock the mutex. The mutex is already locked, so it causes their animation thread to wait until the original one releases the lock, by which time there is no energy left in the resource.

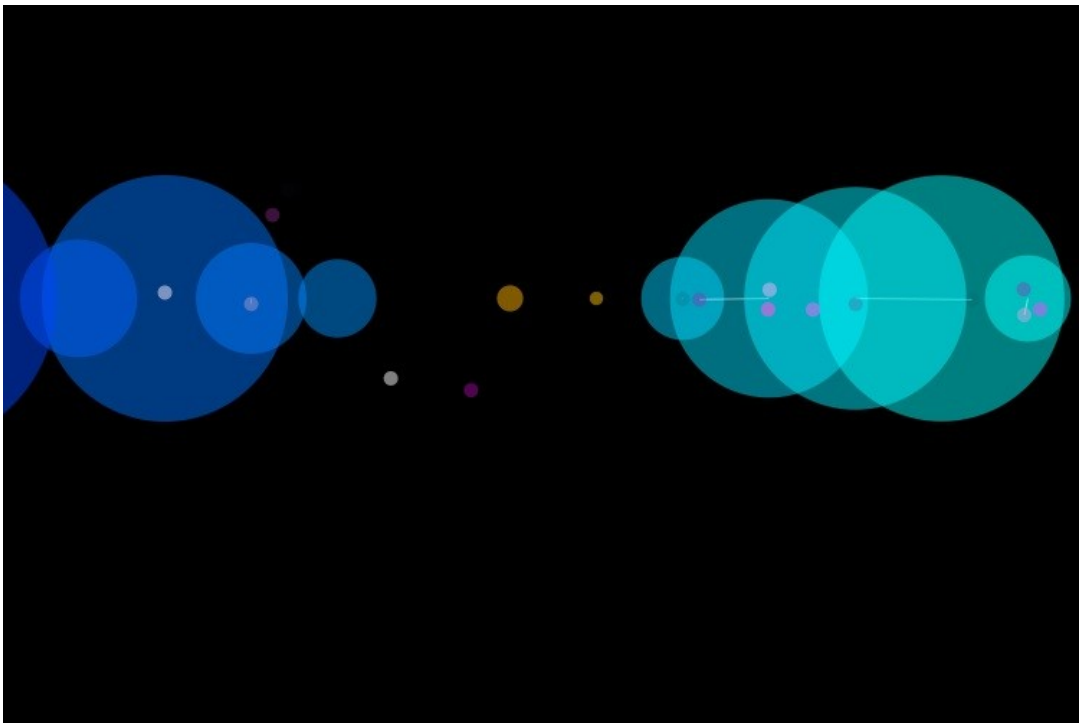

Figure 7. Screenshot of *Limited Resources.* Blue and orange spheres on the x-axis represent resources. Each of the purple/white (consumer) dots orbit the center of the screen, with their update and animation function running on their own thread. When a consumer lands upon a resource, it locks that mutex, and the sphere begins to shrink while emitting a sound at a frequency proportional to the distance the consumer is from the center of the screen. The lock stops the animation of subsequent consumers who land on the same resource, since they also try to lock it.

The system has a human operator who controls it. The operator can add new seeking objects with randomized speed and orbit radius, or increase/decrease the maximum amount of energy a resource is allowed to contain.

Audio is generated when certain events happen. As a seeker consumes a resource, it emits a tone with frequency proportional to its distance from the center of the screen. Because the resources are evenly spaced, the system tends to converge, or tune itself to a semi-harmonic state once it reaches equilibrium. However, because resources can grow large enough to overlap

other resources, it is possible that this will not happen if there are too few consumers. Lastly, when a seeker tries to consume a resource that is locked, a clicking sound is emitted.

There is also something worth noting in the ecosystem-like behavior that happens – certain seeker entities have the 'right' speed so that they tend to be more resilient to changes the operator makes to the system, by far outliving the average new seeker entity that is added. There is also a symbiotic relationship that can occur, or a competitive relationship that happens when two seekers are within a close orbit.

When this piece was presented, numerous comments were made that attributed certain moral tendencies (evil, apathetic) to the seeking entities. This suggests that the relationship among low-level programming elements such as threads are useful not only for implementing a certain natural behavior, but also for revealing and dissecting higher level constructs such as ethics.

4.2.4 Discussion

The piece presented in this paper can be analyzed as an abstracted simulation of some system (e.g. Darwinistic survival,) or as an implementation of an abstract thread system that is visualized and sonified. The initial intent was the latter, but as development occurred and the analogy became clearer, the former became more and more important. An

abstract system 'wants' to have something imposed upon it, or rather, we as observers want to see something concrete, and we do this by imposing our analogies upon the system. This in turn makes its development dynamic. One conclusion I have come to from this experiment is that the composer needs to be sensitive to this dynamism when talking about using 'abstracted' design patterns and data structures for musical composition.

Limited Resources is available as an Objective C++ XCode project via an svn module at http://stria.dartmouth.edu/kdrepos/chinentest/deadlock/.

4.3 Dict

Dict is an audiovisual work that looks at the meaning of words. It uses an online dictionary, (the linux *dict* tool,) to look up the words of a seed sentence. A speech synthesizer is used to vocalize the sentence. The sentence is shown on the screen through a GUI. Words are highlighted as they are read, in a similar fashion to karaoke singing. When the entire sentence is read, the program looks up a random word that tends to be longer than the others, and replaces it with its dictionary definition. There are four 'readers' that can be running simultaneously on independent sentences. The user can control what the seed sentence is, and when each voice should enter. A typical performance might start with a sentence like "this sentence has meaning, somewhere," or "colorless green ideas sleep furiously." Note that the latter sentence, composed by Noam Chomsky [73] is grammatically correct, but

nonsensical.

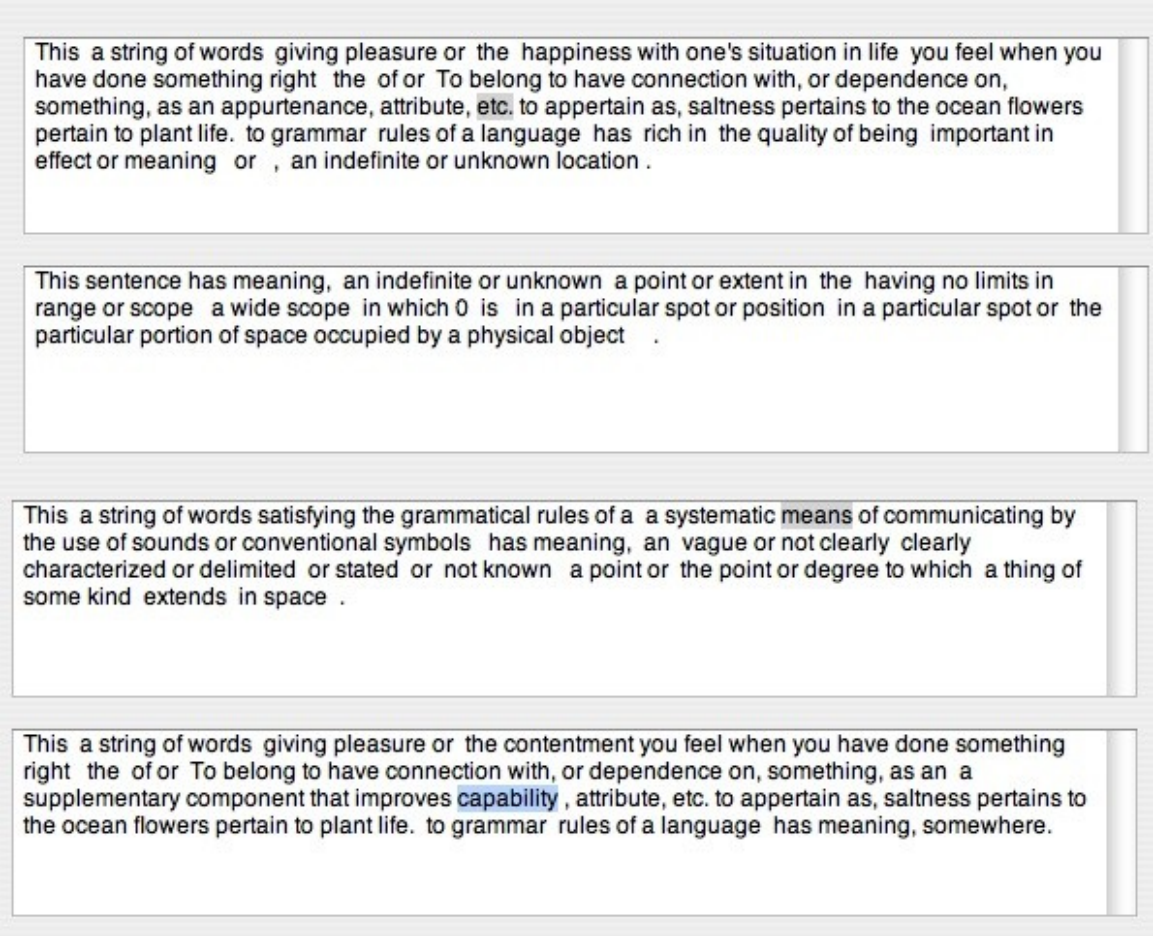

Figure 8. Screenshot of *Dict*. The resultant sentences after several minutes of running *Dict* on the sentence "This sentence has meaning, somewhere." All four boxes started with the same sentence. Since a random word gets looked up and replaced, the sentences diverge from each other in their structure. However, their semantical differences diverge slower, as dictionary definitions often preserve semantics. Nonsensical or awkward artifacts can happen when they differ, e.g., when there are multiple definitions. As an example, the original final word in this sentence is 'somewhere,' and all the sentence endings still have something to do with position.

Since the dictionary definition of a word will contain many others, the sentences will be ever-expanding. The performance needs to be ended by quitting the program while it is running.

The dictionary lookup of individual words ignores structural elements in the

sentence such as context and idioms. *Dict* shows how important these structural yet non-grammatical elements are to our comprehension of language. Instead of trying to create sentences using generative grammars, *Dict* takes an approach to semantics using the power of reference in language to create structure.

The synthesis engine uses Apple's speech synthesis library. It allows for a range of voices, from human speech to song, to noisy and alien voices. The piece has been performed with the song voices. The song is used to create canons between the four voices. They become less cannon-like as the sentence structure begins to differ in later iterations.

The software is available as an Objective C++ XCode project via an svn module at http://stria.dartmouth.edu/kdrepos/chinentest/speechgrammar.

5 CONCLUSION

The concepts presented in this thesis, their relationship to the surveyed work, and the pieces presented in this thesis have shown how the relationship between programming and music might be more closely connected than it seems at a first glance. Current trends in computer science, architecture, design, and media art suggest that there is a reason to study the internals of software, despite the availability of tools that allow development while hiding the software aspects. The concepts presented in chapter 3 focus on several of these reasons for the creation of music. The programming concepts of hierarchy, reference, and synchronicity were not invented for programming. Rather, they exist in programing because humans need these concepts to create and understand phenomena, including music and natural systems.

This work opens a number of conceptual doors for future work. The creation of GUIs that make the actual software process transparent holds potential for art and design. Writing these pieces, *Tree Music* in particular, has been one of the most challenging programming exercises I have done. I have come to look at them as a sort of programming etudes that are both musically and technically challenging. Additionally, I feel that writing these pieces has

given me a clarity on the nature software and programming. As such, this work was very personal. Some aspects of this music allow non-technical audiences to appreciate computer processes, but there is much that could be done to extend this music towards a society that has an everyday relationship with software. Breaking down the mental barrier of software as the unnatural system will be an important topic with regards to the software and music that follow.

Bibliography

[1] Bond, Gregory W., "Software as art," *Commun. ACM*, vol. 48, no. 8, pp. 118--124, August 2005

[2] Knuth, Donald E., "Computer programming as an art," *Commun. ACM*, vol. 17, no. 12, pp. 667--673, December 1974

[3] Puckette, Miller, "Pure Data: another integrated computer music environment," in *in Proceedings, International Computer Music Conference*, pp. 37--41, 1996

[4] Puckette, Miller, "Combining Event and Signal Processing in the MAX Graphical Programming Environment," *Computer Music Journal*, vol. 15, no. 3, pp. 68--77, 1991

[5] Loy, Gareth and Abbott, Curtis, "Programming languages for computer music synthesis, performance, and composition," *ACM Comput. Surv.*, vol. 17, no. 2, pp. 235--265, 1985

[6] Boulanger, R., *The Csound Book: Perspectives in Software Synthesis, Sound Design, Signal Processing,and Programming*, The MIT Press, 2000

[7] Garton, B. and Topper, D., "RTcmix--Using CMIX in Real Time," in *Proceedings of the 1997 International Computer Music Conference*, pp. 399--402, 1997

[8] Moore, F. R., *Elements of computer music*, Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1990

[9] Mccartney, James, "Rethinking the Computer Music Language: SuperCollider," *Comput. Music J.*, vol. 26, no. 4, pp. 61--68, 2002

[10] Wang, Ge, "On-the-fly Programming: Using Code as an Expressive Musical Instrument," in *In Proceedings of the International Conference on New Interfaces for Musical Expression*, pp. 138--143, 2004

[11] Collins, N. and Mclean, A, and Rohrhuber, J and Adrian, W, "Live coding in laptop performance," *Organised Sound*, vol. 8, no. 03, pp. 321--330, 2003

[12] Bond, Gregory W., "Software as art," *Commun. ACM*, vol. 48, no. 8, pp. 118--124, August 2005

[13] Cramer, F. and Gabriel, U., "Software Art," *American Book Review, issue" Codeworks"(Alan Sondheim, ed.),* Sept 2001

[14] "Aesthetic Computing Manifesto," *Leonardo*, vol. 36, no. 4, pp. 255--256, August 2003

[15] Alexander, Christopher and Ishikawa, Sara and Silverstein, Murray, *A Pattern Language: Towns, Buildings, Construction (Center for Environmental Structure Series)*, Oxford University Press, 1978

[16] Maeda, J., *Design by numbers*, , MIT Press, 1999

[17] Polansky, Larry and Burk, Phil and Rosenboom, David, "HMSL (Hierarchical Music Specification Language): A Theoretical Overview," *Perspectives of New Music*, vol. 28, no. 2, pp. 136--178, 1990

[18] Taube, Heinrich, "Common Music: A Music Composition Language in Common Lisp and CLOS," *Computer Music Journal*, vol. 15, no. 2, pp. 21--32, 1991

[19] Cascone, Kim, "MIT Press Journals - Computer Music Journal - Citation," *Computer Music Journal*, vol. 24, no. 4, pp. 12--18,

[20] Kendall, Gary S., "Visualization by Ear: Auditory Imagery for Scientific Visualization and Virtual Reality," *Computer Music Journal*, vol. 15, no. 4, pp. 70--73, 1991

[21] Oram, Andy and Wilson, Greg, *Beautiful Code: Leading Programmers Explain How They Think (Theory in Practice (O'Reilly))*, O'Reilly Media, Inc., 2007

[22] Knuth, Donald E. , "Computer programming as an art," *Commun. ACM*, vol. 17, no. 12, pp. 667--673, December 1974

[23] Knuth, Donald E., "Mathematical Typography," *Bulletin Of The American Mathematical Society*, vol. 1, no. 2, pp., March 1979

[24] Knuth, Donald E., *Art of Computer Programming, Volume 1: Fundamental Algorithms*, Third edition, Addison-Wesley Professional, 1997

[25] Kemighan, B. W. and Plauger, P. J. , *The elements of programming style*, Computing Mcgraw-Hill, 1974

[26] Stefik, Mark and Bobrow, Daniel, "Object-oriented programming:

Themes and variations," *AI Mag.*, vol. 6, no. 4, pp. 40--62, 1986

[27] Alexander, Christopher W., *Notes on the Synthesis of Form (Harvard Paperbacks)*, Harvard University Press, 1970

[28] Gamma, Erich and Helm, Richard and Johnson, Ralph and Vlissides, John, *Design Patterns*, Addison-Wesley Professional, 1995

[29] Loy, Gareth , "Composing with computers: a survey of some compositional formalisms and music programming languages," in *Current Directions in Computer Music Research,* pp. 291--396, 1989

[30] Roads, C., "Artificial Intelligence and Music," *Computer Music Journal*, vol. 4, no. 2, pp. 13--25, 1980

[31] Straus, Joseph N., *Introduction to Post-Tonal Theory (2nd Edition)*, Prentice Hall, 1999

[32] Bailey, K., "Webern's Opus 21: Creativity in Tradition," *Journal of Musicology*, vol. 2, no. 2, pp. 184--195, 1983

[33] Reich, S. , "Music as a gradual process," *Esthetics Contemporary*, pp. 302, 1989

[34] Xenakis, Iannis , "The Crisis of Serial Music," *Gravesaner Blätter* , vol. 1, 1965

[35] Xenakis, Iannis, *Formalized Music: Thought and Mathematics in Composition (Harmonologia Series, No 6)*, 2nd, vol., Pendragon Pr, 2001

[36] Xenakis, I., "Free stochastic music from the computer," *Gravesaner Blätter*, vol. 26, 1965

[37] Schoenberg, Arnold , Strang, F. and Stein, L., *Fundamentals of Music Composition*, Faber and Faber, 1967

[38] Dahlstedt, P., *Sounds Unheard of: Evolutionary Algorithms as Creative Tools for the Contemporary Composer*, Ph.D. Thesis, Chalmers tekniska högsk., 2004

[39] Bown, O. and Lexer, S. , "Continuous-time recurrent neural networks for generative and interactive musical performance," *Lecture Notes in Computer Science*, vol. 3907, pp. 652, 2006

[40] Chinen, M. and Osaka, N. , "Genesynth: Noise Band-Based Genetic

Algorithm Analysis/Synthesis Framework," in *Proceedings of the ICMC 2007 International Computer Music Conference*, Copenhagen, Denmark 2007

[41] Bown, O. and Wiggins, G. A. , "On the Meaning of Life (in Artificial Life Approaches to Music)," in *Proceedings of the 4th International Joint Workshop on Computational Creativity*, 2007

[42] Blackwell, T. M. and Bentley, P., "Improvised music with swarms," in *CEC '02: Proceedings of the Evolutionary Computation on 2002. CEC '02. Proceedings of the 2002 Congress*, pp. 1462--1467, 2002

[43] Cope, D., *Experiments in Musical Intelligence*, AR Editions, 1996

[44] Dunn, D., "Nature, Sound Art, and the Sacred," *The Book of Music & Nature*, pp. 95--107, Wesleyan University Press, 2001

[45] Di Scipio, A., "Systems of Embers, Dust, and Clouds: Observations after Xenakis and Brün," *Computer Music Journal*, vol. 26, no. 1, pp. 22--32, 2002

[46] Cascone, Kim, "MIT Press Journals - Computer Music Journal - Citation," *Computer Music Journal*, vol. 24, no. 4, pp. 12--18, 2000

[47] Polansky, Larry and Erbe, Tom, "Spectral Mutation in Soundhack," *Computer Music Journal*, vol. 20, no. 1, pp. 92--101, 1996

[48] Armstrong, N., *An Enactive Approach to Digital Musical Instrument Design*, Ph.D. Thesis, Princeton University, 2006

[49] Collins, N.. and Mclean, A. . and Rohrhuber, J. and Adrian, W., "Live coding in laptop performance," *Organised Sound*, vol. 8, no. 3, pp. 321--330, 2003

[50] Sorensen, A. and Brown, A. R. , "aa-cell in Practice: An approach to musical live coding," in *Proceedings of the International Computer Music Conference*, Copenhagen, Denmark, August 2007

[51] Trueman, Daniel and Cook, Perry and Smallwood, Scott and Wang, Ge , "PLOrk: The Princeton Laptop Orchestra, Year 1," *Proceedings of the International Computer Music Conference*, New Orleans, LA, August 2006

[52] Ames, Charles , "Concurrence," *Journal of New Music Research*, vol. 17, no. 1, pp. 3--24, 1988

[53] Ames, Charles, "A Catalog of Statistical Distributions: Techniques for Transforming Random, Determinate and Chaotic Sequences," *Leonardo Music Journal*, vol. 1, no. 1, pp. 55--70, 1991

[54] Meneghini, M., "An Analysis of the Compositional Techniques in John Chowning's Stria," *Computer Music Journal*, vol. 31, no. 3, pp. 26--37, 2007

[55] Didkovsky, Nick, "'Lottery': Toward a Unified Rational Strategy for Cooperative Music-Making," *Leonardo Music Journal*, vol. 2, no. 1, pp. 3--12, 1992

[56] Gresham-Lancaster, Scot, "The Aesthetics and History of the Hub: The Effects of Changing Technology on Network Computer Music," *Leonardo Music Journal*, vol. 8, pp. 39-44, 1988

[57] Alexander, Christopher, *The Timeless Way of Building*, New York: Oxford University Press, 1979

[58] Maeda, John, *Creative Code: Aesthetics + Computation*, Thames & Hudson, 2004

[59] Bly, Sara, "Presenting information in sound," in *Proceedings of the 1982 conference on Human factors in computing systems*, pp. 371--375, 1982

[60] Whitelaw, Mitchell, "Hearing Pure Data: Aesthetics and Ideals of Data-Sound," pp. 45--54, September 2004

[61] Bly, Sara A., "Sound and computer information presentation," Ph.D. Thesis, University of California, Davis, 1982

[62] Vickers, Paul and Alty, James L., "Towards some Organising Principles for Musical Program Auralisations," in *in International Conference on Auditory Display*, pp. 1--4, 1998

[63] Vickers, Paul, "CAITLIN: A Musical Program Auralisation Tool to Assist Novice Programmers with Debugging," in *in Proc. Third International Conference on Auditory Display,* Xerox PARC, Palo Alto, CA 94304, pp. 17--24, 1996

[64] Tenney, J., *Meta+ Hodos and META Meta+ Hodos*, Frog Peak Music, 1986

[65] Papoulis, Athanasios, *Probability, Random Variables and Stochastic*

Processes, McGraw-Hill Companies, 1991

[66] Roads, Curtis, *Microsound*, The MIT Press, 2004

[67] Sipser, Michael, *Introduction to the Theory of Computation, Second Edition*, Course Technology, 2005

[68] Cooley, J. W. and Tukey, J. W., "An algorithm for the machine calculation of complex Fourier series," *Mathematics of computation*, vol. 19, no. 90, pp. 297--301, 1965

[69] Dembski, William A., "Randomness By Design," *Noûs*, vol. 25, no. 1, pp. 75--106, 1991

[70] Russell, Stuart J. and Norvig, Peter, *Artificial Intelligence: A Modern Approach (2nd Edition)*, Prentice Hall, 2002

[71] Turing, A. M., "On computable numbers: With an application to the entscheidungsproblem," in *Proceedings of the London Mathematical Society*, , , 230--265, 1936

[72] Gödel, K., Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I," *Monatshefte für Mathematik*, vol. 38, no. 1, pp. 173--198, 1931

[73] Chomsky, Noam , *Syntactic Structures*, 2nd ed., Walter de Gruyter, 2002