

Applications of Generative String-Substitution Systems in Computer Music

Roger Luke DuBois

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Musical Arts
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY
2003

© 2003
Roger Luke DuBois
All Rights Reserved

ABSTRACT

Applications of Generative String-Substitution Systems in Computer Music

Roger Luke DuBois

The purpose of this dissertation is to create and explore potential taxonomies for using algorithmic string-substitution systems in the generation of music. The focus of the author's research is on using a specific category of string rewriting systems (called Lindenmayer, or L-systems) to generate musical material based on a musical primer provided by a live musician or musicians. The author explores and describes a variety of possible composing methodologies whereby a computer can generate, in real time, appropriate accompanying music and signal processing to a live performer. By experimenting with different taxonomies of mapping source material (live musical input) to accompanying processes (provided by the computer), an extensive system for generating a varied, yet systematically cohesive, corpus of musical work can be achieved. A series of short compositions based on this string-substitution process are included as applications of this system.

Table of Contents

List of Figures	ii
Acknowledgements	iv
Dedication	v
Preface	vi
Applications of Generative String-Substitution Systems in Computer Music	
1. Introduction	1
2. Lindenmayer Systems	7
Overview of L-systems	9
Turtle graphics and branching	13
Lindenmayer Systems and music: prior art	18
3. Mapping and Rewriting Schemes	25
Event-literal symbolic mapping	28
Spatial symbolic mapping	34
Static-length L-systems	38
Parametric symbolic mapping (metadata)	41
4. Interactive Performance Using L-systems	53
Overview of <i>Scheherazade</i>	54
Symbolic encoding of musical information	59
L-systems as transfer functions	62
L-systems as symbolic filters	64
Figurative encoding	68
Branching as polyphonic accompaniment	71
Parametric parsing	74
5. Conclusion	79
Appendix: source code examples	82
jit.linden.c	83
max.jit.turtle.c	90
jit.lindenpoly.c	94
Bibliography	98
Scores	102
<i>Growing Pains</i>	103
<i>Repeat After Me</i>	128
<i>Biology</i>	146

List of Figures

Chapter 2	
2.1 (a simple L-system definition)	11
2.2 (generations 0-5 of the L-system in 2.1)	11
2.3 (simple Turtle Graphics instruction set)	14
2.4 (an L-system definition of the quadratic Koch island)	15
2.5 (generations 0-3 of the Koch island as graphics)	15
2.6 (an L-system definition of Luke's hypothetical fern)	16
2.7 (generations 0-5 of Luke's hypothetical fern as graphics)	17
2.8 (generation 1 of the Koch island realized as music)	19
2.9 (flowchart of Prusinkiewicz' musical interpreter)	20
2.10 (flowchart of <i>Scheherazade</i>)	22
Chapter 3	
3.1 (common HCI mappings for personal computers)	27
3.2 (a simple L-system definition)	29
3.3 (generation 8 of the L-system in 3.2)	29
3.4 (generation 8 realized as music)	29
3.5 (rhythmic palette of 3.4)	30
3.6 (an L-system definition of Luke's hypothetical fern)	30
3.7 (generation 5 of Luke's hypothetical fern)	30-31
3.8 (pitch mapping chart for <i>Growing Pains</i>)	32
3.9 (motif from <i>Growing Pains</i> in sections A and F)	32
3.10 (motif from <i>Growing Pains</i> in sections B and H)	33
3.11 (motif from <i>Growing Pains</i> as graphics)	33
3.12 (a context-dependent L-system)	35
3.13 (generations 0-5 of the L-system in 3.12)	35
3.14 (generations 0-5 of the L-system in 3.12 as chords)	36
3.15 (the music in 3.14 re-written on two staff systems)	37
3.16 (generations 0-5 of the L-system in 3.12 as a melody)	38
3.17 (the Sierpinski triangle defined as an L-system)	39
3.18 (the Sierpinski triangle as graphics)	39
3.19 (the Sierpinski triangle as music)	41
3.20 (an L-system with a six-symbol alphabet)	42
3.21 (generations 0-5 of the L-system in 3.20)	42-43
3.22 (a simple musical parser)	43
3.23 (generations 0-5 of the L-system in 3.20 as music)	44
3.24 (a parametric musical parser)	46
3.25 (generation 5 of the L-system in 3.20 as music)	48
3.26 (statistical breakdown of the L-system in 3.20)	50
Chapter 4	
4.1 (flowchart of <i>Scheherazade</i>)	54
4.2 ('The Star Spangled Banner' encoded as a numerical sequence)	61
4.3 (pc/octave breakdown of 4.2)	61
4.4 (a symbolic transfer function as an L-system)	62
4.5 (a more complex symbolic transfer function)	63

4.6 (chromatic scale played through the function in 4.5)	63
4.7 (melodic fragment)	65
4.8 (selective symbolic transfer function)	65
4.9 (melody in 4.7 transformed by the L-system in 4.8)	66
4.10 (an L-system with a six-symbol alphabet)	68
4.11 (an instruction set for the L-system in 4.10)	68
4.12 (melody in 4.7 transformed by the L-system in 4.10)	70
4.13 (impulse response of the L-system in 4.10)	71
4.14 (an L-system with branching and a three-symbol axiom)	72
4.15 (generation 2 of the L-system in 4.14 as a polyphony graph)	72
4.16 (a musical parser for the L-system in 4.14)	73
4.17 (generation 2 of the L-system in 4.14 as music)	73
4.18 (L-system for the echoes in <i>Repeat After Me</i>)	76
4.19 (L-system for granulation in <i>Repeat After Me</i>)	77

Acknowledgements

I'd like to thank the following people for their generous help and support throughout my career at Columbia: Brad Garton, Fred Lerdahl, and Jonathan Kramer for their help and support on this dissertation and all of my endeavors; Terry Pender, Thanassis Rikakis, Dan Trueman, Douglas Repetto, Kate Hofstetter, Tania Saintil, Douglas Geers, Matthew Suttor, John Halle, David Birchfield, Rozalie Hirs, James Fei, David Topper, and all the faculty, staff, and students of the Computer Music Center, Columbia University, both past and present; David Zicarelli, Joshua Kit Clayton, Jeremy Bernstein, Richard Dudas, Randall Jones, Adam Schabtach, Darwin Grosse, Lilli Hart, and everyone at Cycling'74 involved in the Jitter project, especially Gregory Taylor and Chris Dobrian for taking time from their busy schedules to offer their advice and assistance on this project; Toni Dove, Elliott Sharp, Michael Gordon, Julia Wolfe, Chris Mann, Paul Lansky, Perry Cook, Curtis Bahn, Tomie Hahn, Paul D. Miller, Claude Ghez, Eric Rosenzweig and the staff at Engine27 and Harvestworks for their support and encouragement; Robert Rowe, Dafna Naphtali, Joel Chadabe, Tom Beyer, and the faculty and staff in the Music Technology Department of the Steinhardt School of Education, New York University; Jared Lowder and my graduate students at the School of Visual Arts for their encouragement; Joel Ryan, Frank Balde, and everyone at STEIM, for their support in the Summer of 2000; Anne Gefell at the Columbia University Music Department; Katharyn Yew and Meg Schedel; Stephen Krieger, Paul Feuer, Rachael Finn, Mark McNamara, Josh Druckman, Johnathan Lee, Ken Thomson, and everyone who's been involved with the Freight Elevator Quartet over the past six years; Natacha Diels and Maja Cerar for performing these pieces; the DuBois family; Susan Gladstone, for her love and support.

Dedication

This is for Harriet DuBois, and all who loved her.
I hope where she is, she can hear this.

Preface

In the spring of 2000 I took a break from composition to try my hand as a video artist and installation designer. Working with Mark McNamara, I completed a set of pieces over the following years that attempted to tie interesting aspects of visual information with a relevant and complementary soundtrack using methodologies borrowed from my research into auditory display. The resulting works, starting with SoundScape Navigator, concentrated on using Fourier analysis of interesting sounds to generate virtual visual spaces that corresponded to sound environments. That summer, during an artist residency at STEIM in the Netherlands we experimented with camera-tracking technology to accomplish the opposite effect: allowing a performer to “draw” a sonogram and then see it projected in real-time as the sound was synthesized, as if the performer were painting on a canvas that functioned as a musical instrument. While these experiments were somewhat limited, they offered a fairly interesting system for what Mark and I called “cutting art both ways,” translating visual material into sound and vice versa using exactly the same intermediary algorithm and technology. I found this idea very attractive, but the implementation I was following at the time left much to be desired, mostly because of the directness of the audiovisual translation and the lack of any structural methodology. The simplicity of the system, which made it perfect for installation pieces and movement experiments, made it next to useless when trying to integrate it into a performed piece that had to withstand repeated critical listening and viewing.

The following year I was invited to join the development team at Cycling’74 on a new set of extensions to Max/MSP, an object-oriented programming and development

environment for music, sound, and multimedia that was originally developed at IRCAM, and which for the last decade has served as the closest thing in interactive music systems to a lingua franca. We were given the provisional mandate of developing a set of Max objects to enable the real-time manipulation of video streams. In designing the system, however, a technical Pandora's Box had become opened, as it was quickly established that video was simply a subset (and a very small one, at that) of information that can be stored as a sequence of matrices in computer memory. Once the concept of matrices had been defined, it became our task to extend the paradigm to include support for other types of data that could be constructed as a matrix (including audio, spectral data, musical information wrapped as MIDI, etc.). It was by delving through possible applications for matrices that I stumbled across Lindenmayer and his amazing work, and realized that his systems for string substitution and re-writing, initially designed for the task of visualizing growth algorithms to better understand plant evolution, were exactly what I was looking for to integrate music and graphics in a new and meaningful way.

The bulk of this dissertation concerns itself with some ideas I have about how to make interesting music, and interesting art, using Lindenmayer systems. The writing that follows this preface takes for granted the idea that computer-mediated interactive performance systems, wherein a human is somehow engaged in a performing endeavor with a machine running an algorithm, is an aesthetically valid and worthwhile modus operandi for a composer. At several points in the work I will discuss issues of interactive performance, such as the technical feasibility of various aspects of the system I've designed, the relative merits of different program designs, and the value of efficiency and determinism in developing interactive software that can accept an arbitrary input. At all

times I will be taking as rhetorical the question of whether the medium of interactive performance itself is of any use. Rather than leave that question as completely rhetorical (though it would probably make my life easier), I will attempt to clarify and defend my assumptions here.

Many of the fundamental assumptions about involving computers in the performance process (whether in music, dance, or any other medium where human agency is somehow involved) pivots around two axioms, one of which is self-evident, and other which seems, at face value, completely absurd. The first is that computers can accomplish tasks that are outside the reach of human action. A computer running an audio synthesis algorithm can generate streams of musical information of a speed and complexity that exceeds human performance dexterity. Put simply, my computer can generate notes faster, with better timing, and with a better memory, than your best violinist. While this may be true, there is implicit in that statement the flawed assumption that a performance that is always perfect, no matter how complex, is somehow better than an imperfect performance by a musician. In fact, most of the cutting edge research in computer music over the last decade and a half, from physical modeling synthesis algorithms to style- and performance-modeling using musical grammars, has been focused on making computer performances more human, i.e. making them less “perfect.”

The other assumption revolves around the murky definition of the word *interactive*. My dictionary informs me that the primary definition of the word interactive is “involving the communication or collaboration of people or things.” A second dictionary defines the word interactive as “allowing or involving the exchange of

information or instructions between a person and a machine such as a computer or a television.”

It turns out that people often conflate these two definitions of the term, to imply that human-computer interaction is somehow a collaborative process. The sad truth, though, is that there are very few scenarios in which computers really respond in a natural way to human performance, enough to convince the human participant that what is happening is truly a collaborative endeavor. Most pieces of “interactive” computer music tend to be merely “reactive”, in the sense that the computer “reacts” to actions by the performer. The performer, typically working off of a score or predetermined series of instructions, only “reacts” to the computer insofar as she or he needs to listen to what the computer does to go to the next step in the musical form. Typically this decision is binary; for example, many interactive pieces use score-following systems to have the computer “listen” to the performer, cueing events in the piece (electronic sounds, samples, signal processing effects, etc.) when the performer gets to a certain note or passage. The interaction that occurs on the performer side is often only a confirmation that the system indeed worked. If the computer misses the recognition of a note, the performer will often replay the note, and so on.

A truly interactive experience can be mediated only through creating computer response scenarios that are more fluid and less deterministic than a simple series of cues. On the human side, composition strategies that encourage active listening on the part of the performer (such as cues to mimic an aleatoric musical line from the computer on the performer’s instrument) can create true interplay between performer and machine. Some of the ideas in this paper outline ways to use models to encourage this type of scenario,

though it is up to the imagination of the composer to design the interactions that will make them succeed.

By creating a truly responsive interactive environment that allows extensive musical interplay between performer and computer, a very satisfying musical experience can result. With so-called interactive discourse permeating our workplaces, our environment, and our living spaces, artists must face the challenge of developing interactive art that creates a dialogue rather than a monologue. Only by demonstrating the successful execution of these pieces over and over again can we create interactive experiences that rise above the noise floor of the machines all around us.

R. Luke DuBois

April 2003

1. Introduction

The evolution of music can be described, to some extent, as the evolution of form. The gradual incorporation and rejection of different formalisms by a musical culture gives a listener insight into both the aesthetic and technical concerns of the music, as well as intuitions into the culture at large. The modus operandi of dissecting a body of musical work to determine its formal underpinnings is a staple of historical and theoretical enquiry, regardless of whether the researcher in question is investigating a particular piece of music, a composer, a genre, a historical culture, or a sociological or psychological premise that may unfold in the music. The complexity of these formalisms, combined with their premise (mathematical, psychoacoustic, theological) provides to some degree a functional barometer for what a given group of composers, and by extension their larger culture, were concerned with at the time.

To take the most relevant example for our purposes, the process of incorporating mathematical processes into the making of music has a long and interesting history. While the term algorithmic composition¹ is only appropriate in certain contexts, the process of using mathematical or pseudo-mathematical rules to extrapolate musical form has been part and parcel of the Western musical experience since the development of musical “canon” form in the 15th Century (Maurer, 1999). While not dependent on algorithms in the strict sense (see below), composers from Mozart onwards have employed mathematical systems to explore compositional strategies.

¹ Defined by Adam Alpern as “the process of using some formal procedure to make music with minimal human intervention” (Lee and Alpern 1995).

The term algorithm² is defined as a detailed and unambiguous sequence of actions needed to perform a task in a finite number of steps. While the term is often misinterpreted as to be synonymous with any task performed on a digital computer, it's important to clarify that algorithmic procedures in music (and in everything else) predate the era of thinking machines. Serial and set-theoretic procedures in music, for example, in many cases fulfill the definition for algorithmic composition in the domains in which serial manipulations apply. The fact that the composer retains a variable degree of control over every other aspect of the composition fails to dilute the fact that she or he has, in effect, “surrendered” some part of the compositional process to a set of mathematical rules.

The degree to which these rule sets are arbitrary and whether they retain their initial significance when implemented in constructing a musical surface often provides the dividing line as to whether music is considered “algorithmic” or not. For example, we seldom perceive Western Classical music to be algorithmically construed, despite overwhelming psychoacoustic and theoretical evidence to the contrary (Lerdahl and Jackendoff, 1983, et al). This is largely due to the evolved nature of the musical idiom, where our perceptual systems and the musical idiom itself have adapted in a mutually coherent fashion. Put another way, composers working in the Classical “idiom” were using basic algorithmic procedures regarding harmonic motion, rhythmic patterning, and formal organization implicitly (these rules were embedded in the musical discourse, and as such seldom needed formal justification for their use). Where an extreme break between a musical form and our prevailing musical perception has occurred (e.g. integral

² An English scientific term borrowed from a Greek mispronunciation of the surname of the 9th Century Iranian mathematician Abu Ja'far Muhammad ibn Musa Al-Khwarizmi.

serialism), we are much more likely to construe that music as mathematically induced, often to the point where we lose interest in it as artistic expression and only derive stimulation from the work in an intellectual fashion.

The use of explicit algorithms and mathematical models in the construction of music is, therefore, a risky endeavor at best. With the advent of the digital computer, many composers have readily committed to the incorporation of abstract mathematical models into their music. Lejaren Hiller's *Illiac Suite* (1957), considered, somewhat arbitrarily, as the "first algorithmic piece of music" (Chadabe, 1996), in many ways sets the tone for what has become a slippery slope towards using abstract, non-musical, and somewhat arbitrary sets of information and equations to generate music. Unfortunately, the fact that music can be mathematically derived does not necessarily imply that mathematics makes for good music, using any but the most pedantic evaluation systems.

So what sorts of algorithms make for good music? Theoretical research in music cognition, psychoacoustics, and contemporary music theory strongly suggests that musical events are interpreted by the human listener as hierarchical structures, which are deconstructed on various levels from the musical surface of event streams to the higher-level perception of musical form, achieved through our musical memory. Various analytic models are available for discerning these hierarchies, ranging from the metaphysical (Schenker) to the mathematical (Riemann). One of the most promising (and logical) theories for deriving musical structure as heard by the listener is based on linguistics and generative grammar models (Lerdahl and Jackendoff, 1983, Lerdahl, 2001). To turn the tables for a moment, we might posit that algorithmic strategies that emulate (or, at the very least, take into account) these hierarchical systems (or that

emulate natural hierarchical systems in general) would make good source material for algorithmic composition.

A further complicating factor in much algorithmic composition is the tendency to jettison what one would term culturally cohesive musical informatics (e.g. standard musical forms, common harmonic practice, etc) in favor of arbitrary mathematical-functional systems. If we take the position that excessive abstraction will make a piece “sound” algorithmic, we could point out that this is often a matter of how the algorithmic component of a piece is applied.³ For example, using a series of mathematical ruminations on the Fibonacci series to generate pitched material for piece is not a priori an overuse of mathematical systems, provided the composer can present this newly auralized information with a degree of clarity by retaining rhythmic, harmonic, and formal models which will be comprehensible to the listener. The intrinsic risk of algorithmic structures within a piece of music is not to misapply them (though “mapping” is always of primary concern) but to use them everywhere at once.

Any number of solutions have been proposed for creating “complete” systems to generate musical material using algorithmic procedures. These range from hyper-generic systemic models (which are often little more than a thin veneer of musical informatics mapped onto existing synthesis or signal processing languages) to highly idiosyncratic compositional frameworks, typically developed by a composer or small research group in response to particular compositional ideas (Oppenheim, 1989, Olafsson, 1988, Hiller and

³ The cases for and against the use of algorithmic procedures in music, particularly electro-acoustic music, have been made many times. For a fairly recent overview of some of the main perspectives see *Computer Music Journal* (Volume 25: 1): “Aesthetics in Computer Music.” The articles by Guy Garnett (“The Aesthetics of Interactive Computer Music”) and Martin Supper (“A Few Remarks on Algorithmic Composition”) cover much of the debate.

Baker, 1957). These systems are never as comprehensive as they seem, and are most often found lacking in their ability to integrate normative music performance practice (live musicians playing live instruments) into their output.

The advent of fully integrated interactive systems for generating music from the computer has the potential to solve a major shortfall of algorithmic composition through the premise of human-machine interaction. Once a human performer is involved, her/his performative actions can themselves be integrated as agents in an algorithmic context (Rowe, 2001, Winkler, 1998). If used correctly, real-time music systems can, to some degree, mitigate against several common by-products of algorithmic procedures in composition.

Rather than proposing an ideal environment for automatic music composition, I'd like to explore some ideas about what one such environment might include, by way of showing one that I've used to make some music in the last year. Rather than picking a particular set of assumptions to use as our algorithmic currency, I'll be exploring an environment based around a particular class of symbolic systems that can be used for algorithmic expression. These systems, called L-systems, which rely, incidentally, on a grammar model of sorts, were never designed to compose music, which makes them a good candidate for our environment insofar as they lack presuppositions particular to the musical idiom. The systems themselves can be used to generate compositional output directly, or to generate input for other algorithmic systems (which may themselves be of the same typology). The recursive utilization of L-systems allows their incorporation into a nested series of meta-systems, allowing for simple control access at multiple levels of the formal hierarchy. Because these systems have very little to do with music, they

can easily be mapped to generate non-musical material that can be linked (through the grammar model) to the music being generated. Since the systems contain within them no particular assumptions about musical idioms, or ways in which to map their output, they demand extensive composer intervention at all levels of the system's hierarchy.⁴

We shall begin by describing L-systems and their intended uses. Following on from a discussion of L-systems per se, we'll investigate other attempts to integrate them into music composition, with a discussion of some particular investigative pitfalls involved in mapping. We'll then look at potential taxonomies for generating interesting musical mappings from the system, and discuss how these particular functions can be implemented in recursion. We'll then continue on to some of the particular morphogenetic attributes of the system and how they can be extrapolated algorithmically to intuit musical factors such as polyphonic density, harmonic navigation, and performer accompaniment. Finally, some key issues with the implementation of this system as a real-time software package for interactive computer music will be discussed, along with a description of one possible package (the *Scheherazade* software).

In order to begin, however, we have to learn a thing or two about plants.

⁴ We could call this type of algorithmic composition semi-automatic, but we won't.

2. Lindenmayer Systems

In 1968 a Dutch botanist named Aristid Lindenmayer published a two-part article in the *Journal of Theoretical Biology* entitled “Mathematical models for cellular interaction in development” (Lindenmayer, 1968). The article proposed an axiomatic system for modeling developmental growth in plants by exploring a class of string-rewriting systems developed by Lindenmayer, akin to but substantially unique from Chomsky grammars (Chomsky, 1956). These classes of string-rewriting systems, called Lindenmayer systems (or L-systems), work on a parallel-substitution string-rewriting model that can successfully simulate morphogenesis.

Before we delve into the systems themselves, we should look for a moment at the underlying philosophy behind Lindenmayer’s research. Lindenmayer’s work derives from the morphogenic theory of *emergence* in multi-cellular life (Taylor, 1992). Put simply, emergence is a formal growth process by which “a collection of interacting units [in a biological entity] acquires qualitatively new properties that cannot be reduced to a simple superposition of individual contributions” (Prusinkiewicz, Hammel, Hanan, and Mech, 1996). Lindenmayer was primarily intrigued by the fact that cellular growth, viewed on any scale, unfolded in a way that defied explanation through the observation of simple local interactions. In higher-level plants, for example, branching structures (apices and nodes) conform to highly complex, self-similar patterns, despite the fact that on a local level there exists no obvious blueprint for such a pattern.¹

¹ Lindenmayer and his colleagues take for granted that such a blueprint does exist as encoded in the genetic makeup of the organism, but that the work involved in unraveling even a simple organism’s genetic information to infer spatial data will remain beyond our means for some time.

A second feature of morphogenesis that Lindenmayer desired to model was the principle that growth in organisms occurs in parallel steps in discrete bursts of time. A plant, for example, will develop from a single stem towards a complex matrix of branches in stages where there is more than one growth development. To thicken the plot even more, these successive stages of organic growth are, themselves, self-similar, and can exhibit formal parallels in spatially diverse locations. Furthermore, Lindenmayer wanted to develop a model which could emulate growth behavior without having to descend to the cellular (or sub-cellular) level of the organism in question, allowing as a starting point the smallest relevant unit of organic growth needed to capture the formal aspects of the organism.

Lindenmayer undertook the task of developing a mathematical formalism that could extrapolate these patterns from a very small set of initial data. Taking as a starting point the assumption that an organism begins to exhibit interesting growth from an initial multi-cellular unit (or state), he developed a set of rules that would operate upon that state in quantum steps, allowing multiple nodes of cellular growth to occur and interact simultaneously. The algorithmic principle upon which he based his work was that of rewriting a *string* that somehow represented the state of an organism at a given growth stage.

String-rewriting models (Thue systems, Chomsky grammars, L-systems) all have in their lineage some form of Turing machine (Turing, 1936), insofar as they incorporate a one-dimensional (and potentially infinite) array of symbols upon which operations are

conducted using a finite instruction set.² String-rewriting systems can often be used to simulate Turing procedures, but should not be confused with Turing machines per se (see Wolfram, 2002). What string-rewriting models add to the table is the mathematical potential for *database amplification*, whereby a simple set of initial data operated upon by a simple set of rules can be used to derive a data set of complexity far exceeding the initial set provided these rules are applied *in recursion*.

Whereas all types of Chomsky grammars function by performing string-rewriting in series (in keeping with Turing's premise of a single active node), Lindenmayer systems perform string-rewriting in parallel, simulating successive static states of evolutionary growth. As a result, they can be used to simulate most classes of fractals (including the basic Mandelbrot and Julia sets, the Sierpinski triangle, and other common spatial models that exhibit self-similarity). In this sense, L-systems exhibit a typology more akin to string-substitution (where the entire string is replaced), rather than simple rewriting.

Since the premise of our algorithmic composition system is based on functional applications of L-systems, we'll begin with a primer on their composition and look at some of their features.

Overview of L-systems

Lindenmayer systems work by performing replacement operations on a string of symbols.

While the actual symbols used are completely irrelevant as far as the system is

² "Symbols" in Turing machines are represented as binary "states" for the machine. There are many on-line resources that explain Turing machines, and their underlying theory of computational modeling (the "Church-Turing Thesis"). A good place to start is to go to www.alanturing.net.

concerned, by convention L-system designers usually stick to letters of the alphabet and other common ASCII symbols (some of which have special significance in specific L-systems interpreters).

Each character space in the string is usually initialized to a null or neutral value, which defines that space in the string as “empty” (i.e. a symbol that has no relevance in the system, often a “space” character or a full stop). At the beginning of the string a starting symbol or set of symbols is inserted.³ This primer string is referred to in L-systems parlance as the *axiom* of the system. The axiom is often denoted by an omega (Ω).

The length (or size) of the string at any given state is always defined as the distance from the starting point of the string to the last non-empty symbol (i.e. the string is always *null-terminated*). Computer applications of L-systems typically demand that the current string is stored in an array of computer memory; rather than resizing the memory every time the string grows, an arbitrarily high limit is imposed on the length of the string, which has nothing to do with the *actual* length of the string at any moment.

Once the axiom is defined, sets of rules are fed into the system to determine how symbols are substituted as the system runs. These rules are called *productions*, and take the form of a *predecessor* symbol and a *successor* string (which can be of any length). If a symbol in the string matches a predecessor, it is replaced by the appropriate successor for that production rule. If a symbol fails to match any of the production predecessors in the system, it is simply copied to the appropriate place in the output string. All symbols in the current string are substituted *simultaneously*. The output (or production) string can

³ By convention, Lindenmayer strings always read left-to-right sequentially.

then be fed back into the system, becoming the input string. The L-system can then run again, resulting in an additional generation of the system.

For example, if our L-system contains the following information (taken from Prusinkiewicz and Lindenmayer, 1990):

\square :	B		
p1:	B	->	A
p2:	A	->	AB

Figure 2.1: a simple L-system definition

Our system starts with a single symbol as its axiom: the letter B. Our production rules dictate that every instance of B in the string is replaced with the A, and every instance of A is replaced with the string AB.

The first five generations (including the axiom) of our string would look like this:

```

B
A
AB
ABA
ABAAB
ABAABABA

```

Figure 2.2: the first five generations of the L-system in Figure 1.1

It's important to reiterate at this point that the string is substituted *all at once*, with each character in the input string checked against all the productions in the system.

Modeled computationally as a set of operations on a numerical array, we compute an L-system by scanning the input array one character at a time, placing its substitution string into an (initially empty) output array. Since the production rules often dictate that a successor contains multiple symbols, the write pointer on the output array is by necessity decoupled from the read pointer on the input array (see Appendix: *jit.linden.c*). Put in simpler terms, the output string is written from scratch, unlike the production rules for

Chomsky grammars, where the string is rewritten *in place*, allowing production rules to influence each other depending on the order in which they are applied. This parallel substitution process is the main attribute of L-systems that makes them useful for modeling growth in organisms.

We can note a few things about this particular L-system to arrive at some additional formal definitions.

- The L-system denoted above is *deterministic*. There is no ambiguity in how it will run given any input string.
- Our L-system has no notion of *context*. “A” substitutes for “B”, regardless of the symbols surrounding it on either side.
- Our L-system shows a remarkable amount of *self-similarity*, reflecting the recursive nature of the substitution algorithm. This self-similarity is a primary feature of Lindenmayer topology that allows L-systems designers to create complex fractal growth structures with a very simple set of rules.
- Our L-system exhibits a general feature of string-substitution algorithms that swap single symbols for words (collections of symbols), which is that they *grow in size* with every generation of the system. In fact, if the production rules interact (i.e. they share symbols), the system will exhibit an exponential growth if any of the production rules have multi-symbol successor strings. String-substitution models that match a multi-symbol predecessor against a smaller successor string fall outside classic Lindenmayer topology, but are commonplace in algorithmic models for dynamic physical systems such as condensation.

- Axiomatic to the prior note is the observation that the size of our L-system will eventually approach an *infinite* limit (this is a well-understood byproduct of systems that perform database amplification). As a correlate, the number of computations required to formulate an output string will increase proportionately with the amount of memory (storage) the string demands. As a result, it's usually appropriate to constrain or predetermine the number of generations the system produces in any given run in order to properly allocate computation time and storage needs.

A Lindenmayer system that is deterministic and substitutes symbols regardless of symbolic context is referred to as a *DOL-system* (*D* for deterministic, *0* to denote that the system is *context-free*). The use of *parametric* Lindenmayer typologies (where the outcome of the algorithm is not always deterministic) and *context-dependent* L-systems will be discussed later as we investigate mapping strategies for algorithmic composition.

Turtle graphics and branching

Because the symbol topology of L-systems, like Chomsky grammars, is semantically agnostic, much of the developmental research into L-systems design has been over how to *interpret* the strings created by the algorithms. Since the system was developed to explore morphogenetic processes, a cohesive visualization scheme was needed to make the strings comprehensible. In early experiments, Lindenmayer translated the alphabet of his strings into geometric primitives, where letters denoted lines of varying length and bearing. Following subsequent experiments by early L-systems designers (Szilard and Quinton, 1979), Przemyslaw Prusinkiewicz eventually settled on

the use of a LOGO turtle interpreter as the best-suited visualization mechanism for L-systems (Prusinkiewicz, 1979).

The meta-language of “turtle graphics,” developed at MIT by artificial intelligence pioneer Seymour Papert and initially implemented in LOGO (Minsky and Papert, 1969), interprets single symbols as instructions for a virtual drawing implement, called a *turtle*. The metaphor of the language is that a virtual “turtle” exists in a two- or three-dimensional space, and can secrete ink from its tail to create a persistent image. The original instructions for the LOGO interpreter were adapted by Prusinkiewicz to be represented by single symbols generated by an L-system.

The initial instruction set for the turtle consisted of four reserved symbols, which were to be generated by L-systems to create visual patterns:

- F Move forward a discrete amount, drawing a straight line from the turtle’s previous location.
- f Move forward a discrete amount without drawing.
- + Turn left a fixed angle, while staying in place (pivot left).
- Turn right a fixed angle, while staying in place (pivot right).

Figure 2.3: Turtle Graphics: simple instruction set, after Prusinkiewicz, 1979

The distance covered by forward motion (d , typically denoted in pixels) and the size of the angle used in each turn (α , defined in angle degrees) were determined at the initializing stage of the LOGO program. The “handedness” of turtle interpreters (i.e. whether + means turn left or right) differs among turtle implementations, and is often user-defined.

For example, to draw a square thirty pixels on a side, we would initialize our turtle to $[d = 30, \alpha = 90]$ and feed it the instructions $F+F+F+F$ (draw, turn left, draw, turn left, draw, turn left, draw).

While turtle graphics interpreters are not the only way to visualize L-systems (we'll look at an example of "direct" string interpretation later when we consider cellular automata algorithms as a class of L-systems), they have the advantage of being computationally efficient and easily implemented.

By incorporating our turtle instruction set into our L-systems alphabet, we can derive complex self-similar images very quickly. For example, the following L-system will generate a quadratic Koch island (Mandelbrot, 1982):

$$\begin{array}{l} \square: \quad F+F+F+F \\ p1: \quad F \quad \rightarrow \quad F+F-F+FF+F+F-F \end{array}$$

Figure 2.4: an L-system definition of the quadratic Koch island

Below are the first four generations of this L-system (including the axiom), drawn using a simple turtle program with 90-degree turning angles. For clarity, the size of the drawing is reduced to 25% of its original size with each generation, and the interior of the generated polygon has been filled with grey.

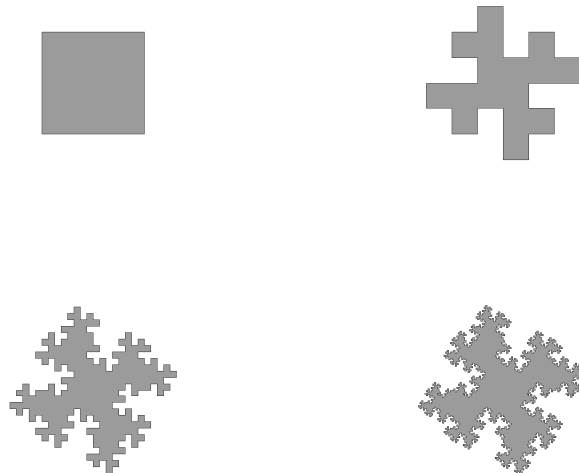


Figure 2.5: generations 0-3 of the Koch island as rendered by Jitter, with $\square=90$ and $d=256, 64, 16,$ and $4,$ respectively

Additional extensions to the turtle interpreter were implemented to help visualize the notion of *branching* in L-systems. In order for L-systems to correctly model the growth of higher plants and other multi-cellular organisms, an extension to the visualization system was needed to accommodate the principle that certain self-similar units within the Lindenmayer string would exist as branches, or offshoots, off of the main trunk of the organism. These branches could be nested, so that multiple “fractal” images could project off of a similar image without forcing the visualized system to retain the shape of a simple polygon.

The symbols “[“ and “]” were added to the turtle lexicon to denote the start and end points of branching structures. In the turtle implementation, this meant that each branch (and each sub-branch) had its own notion of geography. When a branch ends, the turtle’s position and orientation makes a quantum jump back to where it was before the branch started (see Appendix: *max.jit.turtle.c*).

In addition to branching symbols, additional symbols can be added on an ad hoc basis to any given turtle interpreter. In the example below, we’ve added the letter “c” to the turtle instruction set to denote a change of the turtle’s drawing color from among eight different shades of green. This L-system also includes a symbol (“X”) that is integral to the system’s evolution but ignored by the turtle interpreter (a common occurrence in L-systems design).

□:	X		
p1:	X	->	F-[[X]+cX]+F[+FcX]-X
p2:	F	->	FF

Figure 2.6: an L-system definition of Luke’s hypothetical fern

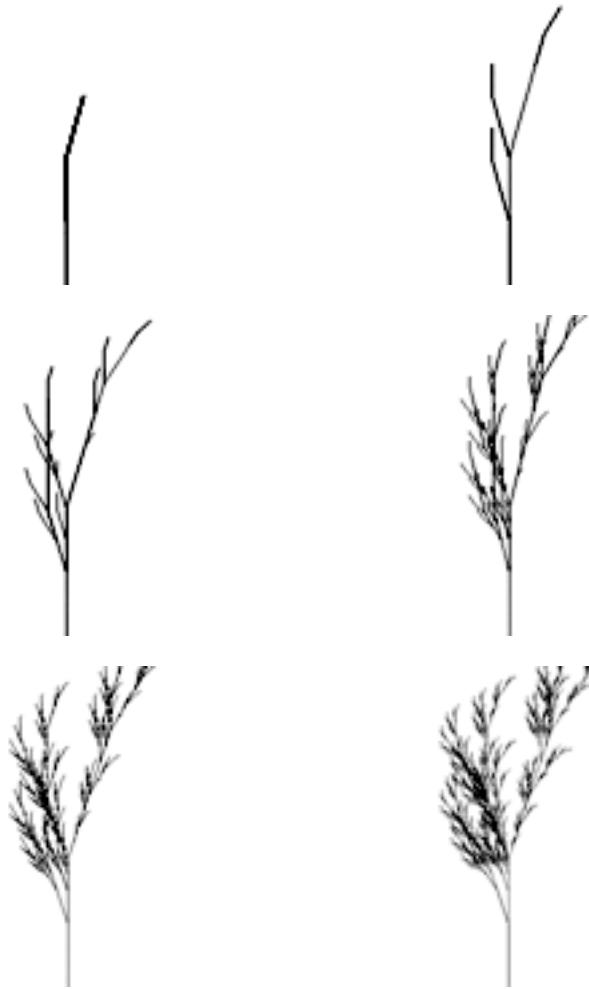


Figure 2.7: generations 0-5 of Luke's hypothetical fern rendered by Jitter ($\square=16$ and $d=96, 48, 24, 12, 6,$ and $2,$ respectively)

By adding branching structures to L-system design, we can create self-similar textures that evolve independently from the overall shape (or trunk) of the system. Further turtle interpreter symbols have been added to plot graphics in 3 dimensions (Prusinkiewicz and Lindenmayer, 1990). While the rest of our investigation will discuss Lindenmayer string interpretation outside of the realm of turtle graphics, spatial metaphors in L-system interpretation (turns, branching, etc.) can be a good source of inspiration when we decide on mapping strategies for music.

Lindenmayer Systems and music: prior art

A number of attempts have been made to integrate L-system methodologies into the composition of music. Most of the compositional work in this area has followed as a natural extension to algorithmic composition using fractals. A number of electronic pieces by Charles Dodge (“Profile”, “Viola Elegy”) implement compositional strategies that trace the geometric progression of fractals mapped onto a microtonal pitch hierarchy. Similarly, Dodge’s piece “Earth’s Magnetic Field” (Dodge, 1987) uses atmospheric magnetism readings as a source data set for the composition; since atmospheric and geological data tends to exhibit substantial self-similarity, this piece reflects some fractal qualities of the source material.

Przemyslaw Prusinkiewicz proposed one potential strategy for composing music using L-systems in a paper given at the 1986 International Computer Music Conference. The paper, entitled “Score Generation with L-systems” (Prusinkiewicz, 1986) has had some impact on a number of composers working with algorithmic composition, notably Gary Lee Nelson, who had been working with fractal music for some time. Their findings have been integrated into a number of software packages, such as David Sharp’s LMUSE software (1995) and the lsys2midi project (Goodall and Watson, 1998).

Prusinkiewicz’s system, and by extension Nelson’s music and the software cited above, works by performing a *spatial mapping* of the L-system output. The L-system output string is first drawn using a turtle interpreter; the resulting geometric shape is then traced in the order in which it was drawn, with the different dimensions of the shape mapped to musical parameters (normally pitch height and amplitude). Each forward movement of the turtle pen along the x axis represents a new note of a fixed metric

duration, with multiple forward movements in the same direction (with no intervening turns) represented as notes of longer duration. A movement along the y axis merely changes the pitch of subsequent notes along a musical scale. A musical evaluation of generation 1 of the Koch island shown above (Fig. 2.5) could look something like this:



Figure 2.8: generation 1 of the Koch island realized using Prusinkiewicz's spatial mapping

Gary Nelson's "Summer Song" (1991) and "Goss" (1993) were both composed using a slightly expanded version of Prusinkiewicz's methodology that took into account angled lines (realized as glissandi) and the possibility of "warping" the shape output by the turtle interpreter using non-standard values of \square . Nelson also realized note duration differently, by computing spatial distance between the vertices that represented the beginning of lines realized as notes.

Other composers have taken similar approaches to spatial mapping of fractal algorithms (including L-systems). Both Stephen Travis Pope and Curtis Roads have incorporated the mapping of fractal systems into their music (Pope, 1987, Roads, 1987). Roads' granular synthesis pieces, in particular, extend the system to the realm of "microsound" (Roads, 2002), where fractal patterns are used to trace the shape of acoustic phenomena in synthesis algorithms, such as the envelopes and waveforms of individual sonic events. By applying such spatial metaphors directly to timbre, Roads has acquired a vocabulary for generating interesting sonic results that take advantage of the potential for self-similarity in L-systems; when mapped to the shape of a waveform, for example, repetitive or fractal patterns become salient as harmonic periodicities in the

signal, creating different spectra through different L-systems. Michael Gogins (1998) has done a number of compositions taking the mapping to fractal patterns to all aspects of a musical composition, effectively creating a self-contained musical vocabulary based entirely on spatial mapping strategies.

Prusinkiewicz also states in his paper that branching in the L-system string could be represented by musical polyphony. Though he doesn't give any examples, one could easily imagine a mapping wherein branches above and below a certain spatial area could be represented as harmony lines to an original melody represented by the "trunk" of the drawn L-system. Both LMUSE and lsys2midi interpret branching as jumps upward and downward to different levels of harmony in musical thirds (similar to chord extensions in jazz); a forward motion four branches away from the main trunk of a system would therefore be realized as a ninth above the main harmony note.

To summarize, Prusinkiewicz's musical interpreter looks something like this:

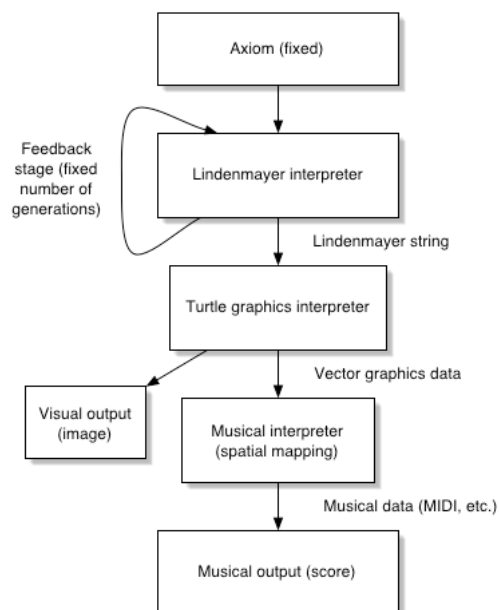


Figure 2.9: Flowchart of the Prusinkiewicz' interpreter

While this form of direct spatial mapping can generate some interesting results, it fails to provide substantial musical functionality in a number of ways:

- By using a fixed axiom, a fixed interpreter, and a finite (and predetermined) number of iterations of the L-system, the result of the string substitution run will always be the same. As a result, the composer is forced to work with a fixed body of data for musical interpretation over the course of a piece. By making the axiom and L-system rules flexible, and by varying the number of passes performed on the string, we can vary the output dynamically to generate more formally interesting results.
- More importantly, the use of a strict spatial mapping effectively negates much of the flexibility inherent in Lindenmayer's string-rewriting model. In effect, sticking a turtle interpreter in the middle of the chain seriously undermines the idea that the symbols in L-system topography are arbitrary, and can represent any type of data. We are forced instead into the presupposition that the L-system string is always tied to a spatial metaphor, with all the limits that metaphor imposes in terms of dimensionality and linear contour. This in turn severely restricts our ability to map the symbols in an expressive (and musically interesting) fashion.

In the following chapters, we'll explore a way of interpreting L-systems musically that avoids some of these pitfalls. We'll attempt to describe a potential system which generalizes many of the points in Prusinkiewicz's methodology; in a number of instances, we'll jettison his ideas (such as reliance on a turtle interpreter) altogether. The basic outline of one possible configuration for a generalized musical system is shown below:

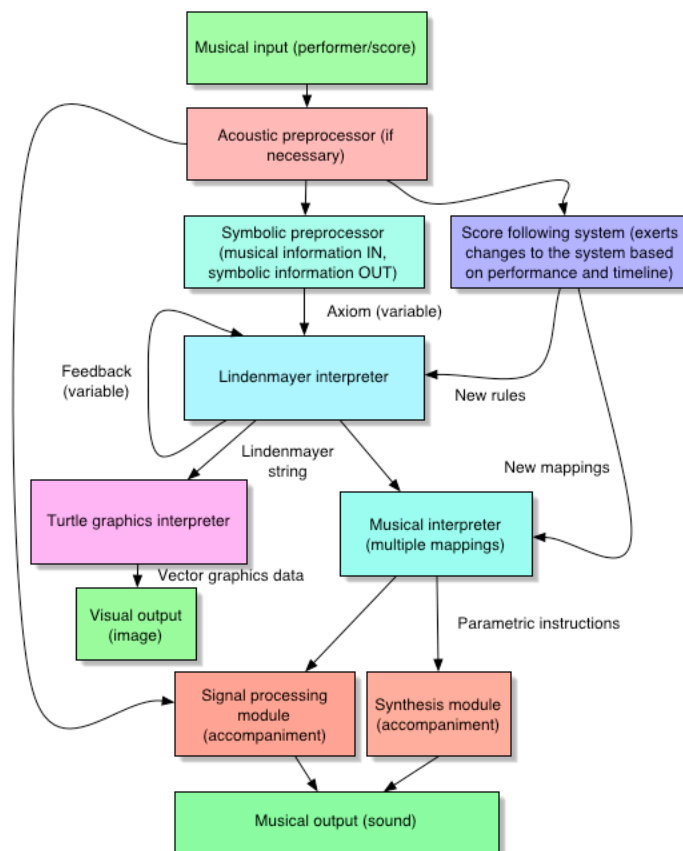


Figure 2.10: A flowchart of a more general system, including decoupled musical and graphical interpreters, a score follower, and the potential for live performer interaction

The system outlined above, which we'll use as a starting point for exploring different interpretation schemes in the next chapter, contains a number of improvements over the previous system in terms of its flexibility. Most notably:

- The turtle graphics interpreter is decoupled from any interaction in the musical process. In fact, we could even omit it entirely. If we still desire a spatial mapping of the L-system, we can introduce rules into the musical interpreter module (now a separate entity) that mimic the graphical behavior of the turtle. This is both more flexible and more efficient: we no longer have to interpret the turtle's moves on a graphical surface to infer musical data; we can derive that data

directly from the Lindenmayer string by interpreting symbols as cogent musical information.

- A score following module is added to the system, allowing the interpretation (both within the Lindenmayer rule set and within the musical mapping system) to change over time, in response to a formal timeline or in reaction to performer input (see below). This eliminates any restrictions on musical output that are asserted by using the same set of rules over and over again throughout a piece.
- Our system replaces a set axiom with symbolic information supplied by a human performer (or, for that matter, a computer performer). A composer can supply the performer with a score, improvisational guidelines, or a set of rules that she or he can then use to input a stream of variable data into the system. As a result, the music generated by the system no longer depends on the Lindenmayer rule set as the sole variable factor. This addition alone allows much more flexibility than before. In addition, the performer can exert control over the score following mechanism, which can change the rules applied to the performer's input.
- The output of the system is no longer constrained to a simple stream of algorithmically generated music. Our network can supply musical accompaniment in the form of synthesized sound, or it can use the L-system output as symbolic instructions to perform manipulations on the performer's input, allowing for a much more creative mapping environment.

In the next chapter, we'll explore some different strategies for encoding and interpreting musical information as symbols for use in L-systems. We'll also explore

different mapping strategies, taking advantage of the flexibility inherent in the grammar model to show some interesting uses of string-substitution as a compositional tool.

3. Mapping and Rewriting Schemes

Now that we've explored some of the basic principles and features of Lindenmayer systems, as well as a few of the ways in which the systems are commonly visualized, we can look into possible ways in which we can use the data from L-system interpreters to generate and interpret musical information.

Throughout this chapter (and into the next) we'll be spending the bulk of the discussion focusing on issues of musical *mapping*. With the incorporation of digital computers into the compositional process, the way in which a set of non-musical information is interpreted in an acoustic context is possibly the most important decision a composer can make in a given piece. Depending on the quantity, dimensionality, and polyphony (in the sense of discrete streams) of the data, the ramifications of how numbers are mapped to musical and sonic output can greatly influence the aesthetic impact of an algorithmically conceived piece of music.¹

For the purposes of our investigation an important distinction needs to be made between “external” and “internal” mapping, or “ergonomic mapping” and “transcoding.” Ergonomic mapping, which we'll describe first to briefly introduce the concept of mapping as a whole, is the process by which physical events are translated into instructions for the computer program. These events occur external to the system and, as a result, need to be treated with very few suppositions as to their raw content. For example, a computer program that claims to deduce the pitches of a musical performance, but only works reliably if the performer plays perfectly in tune at all times, would be next

¹ For a good overview of some of the main issues in mapping in interactive music, see Paradiso, 1999.

to useless, since it makes an assumption about the incoming data stream (the sound generated by the performer) that can't be maintained in all conditions.

Internal mapping, or data transcoding, occurs when a set of data with a known vocabulary (an L-system, for example) is “mapped” onto one or more output parameters. In this case, restrictions can be placed on both the incoming data and the outgoing information in such a way that it is reasonably easy to create deterministic results. While the rest of this chapter will focus on internal mapping schemes, some ideas on external mapping are useful to introduce the subject.

A great deal of the relevant discourse vis-à-vis musically cogent mapping exists in research being done in the field of *human-computer interface* (HCI). HCI researchers and developers, whether working within industry, academia, or from the point-of-view of independent software and hardware developers, are constantly grappling with ways in which ergonomic events generated by a human are to be mapped to corresponding actions by a computer (Johnson, 1997). A number of standard theories for HCI mapping exist, ranging from experimental research done at Xerox PARC to the Human Interface Design (HID) standard developed over the years by Apple Computer. Most current HCI theories manifest themselves as suggested guidelines by which software developers are expected to map common human actions on computer peripherals (mouse clicks, combinations pressed on computer keyboards) to a set of standard actions on the computer. Any reasonably computer-literate person will find a summary table of just a few of these mappings immediately familiar:

Human Task	Computer Response
Double Click (mouse)	Open document; execute command immediately
Click and drag (mouse)	Move selected item to the area indicated
Modifier- O (key)	Open document

Modifier-P (key)	Print document
Modifier-S (key)	Save document
Modifier-Q (key)	Quit current program
Modifier-Z (key)	Undo last action
Modifier-X (key)	Cut selection
Modifier-C (key)	Copy selection
Modifier-V (key)	Paste selection

Figure 3.1: Some common HCI mappings used by most personal computer (PC) operating systems

As the interface increases in complexity, so must the mapping scheme. With the creation of digitally interfaced musical instruments in the 1980's, HCI researchers at digital instrument manufacturers were faced with the task of mapping simple and deterministic human gestures to a sonic output. Most manufacturers, following the path of least resistance, simply copied the physical attributes and layout of traditional Western musical instruments (e.g. pianoforte-style keyboard designs, Boehm fingering layouts for wind instruments). These design and mapping decisions were taken largely to encourage acoustic performers to transition to electronic instruments. With the advent of custom-built musical interfaces and the creation of real-time tools to address musical interfaces (including normal acoustic instruments) in a more flexible manner, the issue of mapping is, once again, a relevant topic for composers working with live instruments.²

² Interestingly, the age of pre-digital electronic instruments saw in many ways the most innovation in ergonomic mapping for the musical performer. Anyone familiar with the theremin, which remains the only widely used instrument ever designed that gives no haptic feedback to the performer, will agree that most digital instrument designers in the last quarter-century have completely failed to think outside the box on this issue. Much of this debate has, however, centered on economic, rather than ergonomic, concerns. The dividing line in analogue synthesis systems between synthesizers with keyboards (e.g. Moog systems), and those that featured control surfaces with arbitrary tunings (e.g. Buchla and Serge systems) is well documented; the fact that the latter sold a fraction as well as the former is often overlooked. See Theberge, 1997 for more on the history of digital instruments and musical controllers.

If we define ergonomic mapping as how we get information into a digital system, we can define transcoding as mapping that occurs within the digital domain. We can now look at some *internal* mapping strategies, with the goal of transcoding information from Lindenmayer systems into musically useful material. What follows are some examples of ways to take the symbolic information contained in a Lindenmayer string and extract a musical texture that somehow retains some of the interesting features of the string as musically coherent phenomena.

Event-literal symbolic mapping

As we mentioned in the last chapter, the typical strategy for generating a musical texture from a Lindenmayer string has been to rely on the spatial information generated from a turtle graphics interpreter to generate a suitable musical mapping. While this can generate interesting musical material, a more rewarding line of enquiry might involve looking into the grammar model itself and finding ways to map the symbols contained in the L-system, treating a graphical representation of the string, if any, as a separate and complementary process.

A simple (and reasonably-straightforward) system for mapping L-systems into musical data is to treat the individual symbols in a Lindenmayer string as discrete musical events. An *event-literal* mapping, where each symbol represents a particular musical datum, is most easily accomplished by treating the string as a temporal *stream*, with the musically mapped symbols following one another in the time domain. This has a number of advantages insofar as it maintains the *linearity*, or perceptual one-dimensionality, of the L-system string.

For an example, we could revisit our first Lindenmayer system, defined in the previous chapter:

□:	B		
p1:	B	->	A
p2:	A	->	AB

Figure 3.2: a simple L-system

The eighth generation of this particular L-system results in the following string:

ABAABABAABAABABAABABAABABAABAAB

Figure 3.3: generation 8 of the L-system in Figure 3.2

Perhaps the simplest conceivable mapping of this L-system into an event-literal stream would be to pick a metric value, and substitute all the instances of the letter ‘A’ with a note at that value, with instances of ‘B’ becoming rests. A realization of this mapping (without superimposing a meter) could yield something like this:



Figure 3.4: generation 8 of the L-system in Figure 3.2, realized as sixteenth notes (‘A’ = note; ‘B’ = rest)

Even with a trivial mapping, some of the features of this L-system become salient when treated as musical information, e.g.:

- ‘B’ never occurs more than once in a row. Hence, our maximum rest value in our musical stream will never exceed the base metric value of the mapping (in this case, a sixteenth note).
- ‘A’ never occurs more than twice in a row. It also never occurs once in a row, if you will, more than once in a row, i.e. the substring ‘BAB’ only occurs when surrounded by the substring ‘AA’ on both sides. As a result, our rhythmic texture is *limited* to two patterns.



Figure 3.5: rhythmic palette of the musical stream in Figure 3.4

- The above will remain true regardless of which generation of the L-system we use to generate the musical stream.

Because of the limited alphabet of our L-system (number of symbols), and the modest scope of our production rules, the string generated by our L-system consists solely of a finite number of local substrings, which remain constant in character and relative frequency. While this is all very interesting, this particular combination of Lindenmayer system and mapping strategy is probably unsuitable for the generation of musical material beyond that of a motivic fragment.³

For a more complex and potentially interesting example of literal mapping, we could examine the Lindenmayer system shown before in Figure 2.6:

□:	X		
p1:	X	->	F-[[X]+cX]+F[+FcX]-X
p2:	F	->	FF

Figure 3.6: the L-system for Luke's hypothetical fern

In this case, the fifth generation of the system yields an incredibly complex string of nearly 32,000 characters in length. The first 2,000 or so are shown below.

```

FFFFFFFFFFFFFFFF-[[FFFFFFFF-[[FFFF-[[FF-[[F-[[[X]+cX]+F[+FcX]-X]+cF-[[X]+cX]+F[+FcX]-X]+FF[+FFcF-
[[X]+cX]+F[+FcX]-X]-F-[[X]+cX]+F[+FcX]-X]+cFF-[[F-[[X]+cX]+F[+FcX]-X]+cF-[[X]+cX]+F[+FcX]-
X]+FF[+FFcF-[[X]+cX]+F[+FcX]-X]-F-[[X]+cX]+F[+FcX]-X]+FFFF[+FFFFcFF-[[F-[[X]+cX]+F[+FcX]-X]+cF-
[[X]+cX]+F[+FcX]-X]+FF[+FFcF-[[X]+cX]+F[+FcX]-X]-F-[[X]+cX]+F[+FcX]-X]+FF-[[F-[[X]+cX]+F[+FcX]-X]+cF-
[[X]+cX]+F[+FcX]-X]+FF[+FFcF-[[X]+cX]+F[+FcX]-X]-F-[[X]+cX]+F[+FcX]-X]+cFFFF-[[FF-[[F-
[[X]+cX]+F[+FcX]-X]+cF-[[X]+cX]+F[+FcX]-X]+FF[+FFcF-[[X]+cX]+F[+FcX]-X]-F-[[X]+cX]+F[+FcX]-X]+cFF-
[[F-[[X]+cX]+F[+FcX]-X]+cF-[[X]+cX]+F[+FcX]-X]+FF[+FFcF-[[X]+cX]+F[+FcX]-X]-F-[[X]+cX]+F[+FcX]-
X]+FFFF[+FFFFcFF-[[F-[[X]+cX]+F[+FcX]-X]+cF-[[X]+cX]+F[+FcX]-X]+FF[+FFcF-[[X]+cX]+F[+FcX]-X]-F-
[[X]+cX]+F[+FcX]-X]+FF-[[F-[[X]+cX]+F[+FcX]-X]+cF-[[X]+cX]+F[+FcX]-X]+FF[+FFcF-[[X]+cX]+F[+FcX]-X]-F-
[[X]+cX]+F[+FcX]-X]+FFFFFFFF[+FFFFFFFFcFFFF-[[FF-[[F-[[X]+cX]+F[+FcX]-X]+cF-[[X]+cX]+F[+FcX]-

```

³ However, we could easily incorporate a formal meta-structure that would *vary* the production rules of this L-system over time, generating a different vocabulary of rhythms as the piece progresses.

X)+FF[+FFcF-[X]+cX]+F[+FcX]-X]-F-[X]+cX)+F[+FcX]-X)+cFF-[F-[X]+cX] +F[+FcX]-X)+cF-[X]+cX)+F[+FcX]-X)+FF[+FFcF-[X]+cX]+F[+FcX]-X]-F-[X]+cX)+F[+FcX]-X)+FFFF[+FFFFcFF-[F-[X]+cX]+F[+FcX]-X)+cF-[X]+cX)+F[+FcX]-X)+FF[+FFcF-[X]+cX]+F[+FcX]-X]-F-[X]+cX)+F[+FcX]-X)-FF-[F-[X]+cX]+F[+FcX]-X)+cF-[X]+cX)+F[+FcX]-X)+FF[+FFcF-[X]+cX]+F[+FcX]-X)-FF-[F-[X]+cX]+F[+FcX]-X)+FF[+FFcF-[X]+cX]+F[+FcX]-X)-F-[X]+cX)+F[+FcX]-X)-FFFF-[FF-[F-[X]+cX]+F[+FcX]-X)+cF-[X]+cX)+F[+FcX]-X)+FF[+FFcF-[X]+cX]+F[+FcX]-X)-F-[X]+cX)+F[+FcX]-X)+cFF-[F-[X]+cX]+F[+FcX]-X)+cF-[X]+cX)+F[+FcX]-X)+FF[+FFcF-[X]+cX]+F[+FcX]-X)-F-[X]+cX)+F[+FcX]-X)+F FFF[+FFFFcFF-[F-[X]+cX]+F[+FcX]-X)+cF-[X]+cX)+F[+FcX]-X)+FF[+FFcF-[X]+cX]+F[+FcX]-X)-F-[X]+cX)+F[+FcX]-X)-FF-[F-[X]+cX]+F[+FcX]-X)+cF-[X]+cX)+F[+FcX]-X)+FF[+FFcF-[X]+cX]+F[+FcX]-X)-F-[X]+cX)+F[+FcX]-X)+cFFFFFFFFFF-[FFFF-[FF-[F-[X]+cX]+F[+FcX]- X]+cF-[X]+cX)+F[+FcX]-X)+FF[+FFcF-[X]+cX]+F[+FcX]-X)-F-[X]+cX)+F[+FcX]-X)+cFF-[F-[X]+cX]+F[+FcX]-X)+cF-[X]+cX)+F[+FcX]-X)+FF[+FFcF-[X]+cX]+F[+FcX]-X)-F-[X]+cX)+F[+FcX]-X)+FFFF[+FFFFcFF-[F-[X]+cX]+F[+FcX]-X)+cF-[X]+cX)+F[+FcX]-X)+FF[+FFcF-[X]+c X]+F[+

Figure 3.7: opening segment of generation 5 of Luke's hypothetical fern

In this example, a number of factors make this a potentially interesting candidate for direct mapping. For one, it has a greatly expanded alphabet consisting of the symbols {F, +, -, [,], c, X}. While this alphabet was obviously selected to make it compatible with a turtle interpreter, we could map this system by creating a transition table of pitches to correspond to each symbol. In the piece *Growing Pains* (see Scores), this string is partially realized by literal mapping, using different symbol-to-pitch mapping sets in different sections of the piece according to the following table. In all sections of the piece, the symbol 'X' is interpreted as a rest, reflecting its use as metadata in the string (it has no effect on the turtle interpreter, either). The letter 'P' is used to denote that a particular pitch is used as a pivot note, instructing the computer that the performer is transitioning to the next section; however, the pivot pitch is always the pitch class represented by 'F' in the subsequent section, and represents that symbol in the mapping in all other respects.

	Section Letter (Symbol Mapping)									
Pitch Class	A	B	C	D	E	F	G	H	I	
C 0	F	[[-	(P)	F	c	[
C#/Db 1]]		
D 2]			+	+]	[+	
D#/Eb 3	(P)	F]			(P)	F		
E 4			-		-	(P)	F		-	
F 5	+	-	+		[+		-	[
F#/Gb 6		+]	+		
G 7	-		(P)	F	c	-	+		c	
G#/Ab 8		(P)	F	[
A 9	c		c	(P)	F	c	-	(P)	F	
A#/Bb 10	[c]	c]	[c]	
B 11										

Figure 3.8: pitch mapping of Lindenmayer symbols in *Growing Pains*

Using this mapping scheme, the literal translation of symbols to pitches exposes some of the interesting features of our L-system. For example, the fifth generation string contains a number of substrings that recur, but unlike our first example, these recurring patterns are followed by a variable amount of transitional material that changes throughout the piece.

One of the most common substrings we find in the L-system for *Growing Pains* is $F-[[X]+cX]+F[+FcX]-X]+$. This seems logical, since (except for the last two symbols) it matches the production successor for the symbol 'X', indicating that 'X' was there in the previous generation. In the first two thousand symbols alone (shown above), it occurs 48 times! This central melodic 'riff' appears with different pitch mappings in the piece, depending on how we have the symbols mapped. In sections A and F, it appears in the score as:



Figure 3.9: the string $F-[[X]+cX]+F[+FcX]-X]+$ in sections A and F

note motif to accompany the three instances of 'F' in the string), and still have it retain its salience when the music and graphics are generated in tandem.

An event-literal mapping can generate interesting musical material with little pre-compositional effort. By using a more complex mapping (where symbols are substituted by chords, rhythmic passages, etc.) a cohesive stream of musical information can be obtained that adequately reflects many of the self-similar features (such as repeating cells and alternating patterns) that are the primary salient features of many Lindenmayer systems. In general, however, we might want to consider incorporating this mapping technique into a more robust body of mapping schemes, which interprets the symbols not as discrete agnostic phenomena, but as pieces of a larger formal context.

Spatial symbolic mapping

Another possible scenario for mapping Lindenmayer strings into musical data is to treat the string itself as a type of musical *space*. The idea is to treat each generation of the L-system as a separate sonic texture, with some sort of system in place to realize the texture and interpret between *successive generations* of the grammar model (as opposed to our event-literal examples, which focus only on one particular generation of an L-system at a time).

An easy way to visualize how this would work is to take a class of Lindenmayer systems called *context-dependent*, or 2L systems. These systems work in the same way as context-free models, with the added caveat that substitution of a predecessor symbol for a particular substitution string depends on the symbols surrounding that predecessor. For example:

□:	BAB	
p1:	AB ->	BA
p2:	A<A>B ->	AB
p3:	A<A>A ->	B
p4:	B<A>* ->	AA

Figure 3.12: a context-dependent L-system

This L-system defines the following production rules, given the axiom ‘BAB’. If the symbol ‘B’ is found in the string, it is replaced with the string ‘BA’ *only if* the symbol immediately preceding the ‘B’ is ‘A’, and the symbol following the ‘B’ is also ‘B’. The symbol ‘A’ is replaced depending on its context as well: if the ‘A’ is preceded by another ‘A’, it is substituted by the string ‘AB’ if the symbol after it is ‘B’; if the following symbol is another ‘A’ (i.e. three ‘A’s in a row), then it is replaced by a ‘B’. Finally, a symbol ‘A’ preceded by a ‘B’ and followed by *anything* (denoted by the wildcard symbol, ‘*’) is replaced with the string ‘AA’.⁵ The first five generations of this L-system (including the axiom) look like this:

```

BAB
BAAB
BAAABB
BAABABBAB
BAAABBAABABAAB
BAABABBABAAABBAABAAABB

```

Figure 3.13: the first five generations of the L-system in Figure 3.12

We could attach a simple musical mapping to this string as follows: let every symbol represent a semitone above a starting pitch (say C 3, with the first symbol in the string representing that note as a root of sorts). The letter ‘B’ represents a note at that

⁵ It’s important to note that the *ordering* of the production rules is important when using L-systems where each symbol can match more than one production rule. If a rule with context wildcards occurs earlier in the list of rules than a rule with a context limitation, the first rule will always match, as it will be checked first.

pitch height, with the letter ‘A’ functioning as a placeholder symbol to indicate the relative position of the next ‘B’. Out of this grammar model we’d get something like this for generations 0-5:



Figure 3.14: the first five generations of the L-system in Figure 3.12, mapped as block chords

Some of the interesting features of this L-system become musically salient even with this simple mapping. For example, our L-system is constructed in such a way that points of stability emerge fairly quickly: the first two symbols will always be a ‘BA’; by generation 2, the sixth symbol will always be a ‘B’, with neighbors alternating between a ‘B’ on the left and an ‘A’ on the right, and vice versa. Each successive generation adds another stable point where a ‘B’ is present (the ninth symbol for generation 3, the fourteenth symbol for generation 4, etc.).

With our simple musical mapping, we immediately get results out of this feature in the form of a static underlying harmony that increases in complexity and persists throughout the succession of chords. This becomes obvious if we split our texture into two staff systems, with the persistent notes on the lower system and the variable notes on the top:

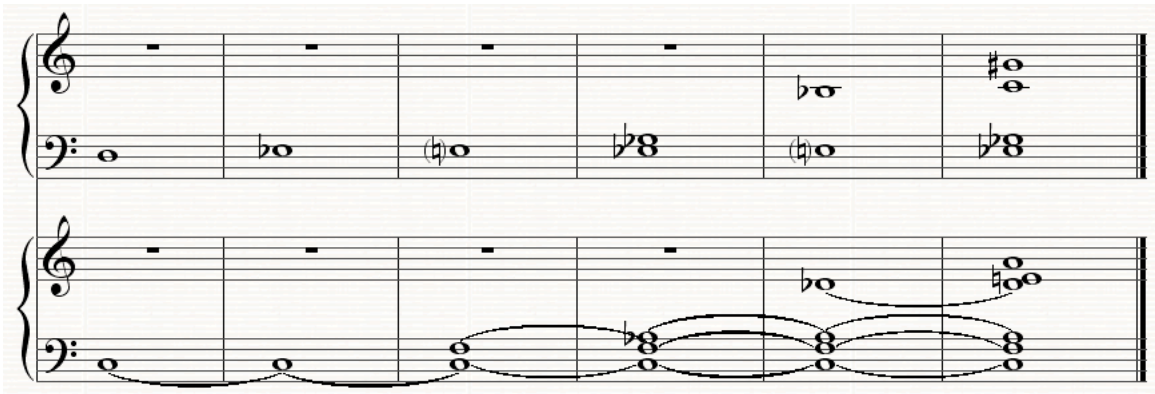


Figure 3.15: our harmonic L-system mapping broken up into two staff systems. The lower system only shows those notes that persist from generation to generation.

Looking at the music above we can clearly see a constant harmony emerge from our mapping of the system. By measure five, curiously, we've generated what one could describe as a Db major 7th chord (with the leading-tone in the bass). Similarly, other repeating figures become apparent, such as the alternating presence of Eb (and later Ab) and E with each generation. As the string expands, more of these persistent and alternating features will appear. If we consider temporal persistence and periodic alteration as musically salient phenomena that we want to emphasize, we can easily develop a high-level parser to find static and repeating symbols between generations of the L-system. For example, this parser could extract this data for use in decisions of orchestration priority, and would begin to mimic some of the basic features of our cognitive listening apparatus (Bregman, 1992).

If we unravel the musical texture to incorporate a time component, we can treat each generation of the L-system as a discrete, self-expanding musical phrase. An example mapping would be to treat each symbol as indicative of pitch height as before, but add in a rhythmic element where each symbol (regardless of whether it represents a note) advances the timeframe of the musical phrase by a set metric value. If we wrap the

pitch material around to a single octave (by performing a modulo-12 operation on the pitches) and lengthen the note durations so as to remove rests caused by the symbol ‘A’, we could get something like this:



Figure 3.16: the first five generations of the L-system in Figure 3.2, mapped melodically. Phrase marks represent each generation of the L-system.

In addition to the persistent pitches occurring with their expected regularity, the rhythmic mapping allows us to successfully represent the alternating patterns of groups of ‘A’ symbols in the string (represented as longer note values).

Static-length L-systems

A related and similarly interesting mapping scheme that is worth visiting concerns Lindenmayer strings that remain of a static length from generation to generation. While the L-system grammar was developed, as we have seen, to capitalize on principles of database amplification to show development in stages, it’s trivial enough to develop an L-system where the axiom sets the size for successive generations of the grammar model.

A set of production rules where each predecessor symbol is substituted by a single successor symbol will create a static-length L-system. L-systems that don’t grow can be considered as a class of *cellular automata* in one dimension, and exhibit many of the characteristics of these algorithms (Bossomaier and Green, 1998). With CA algorithms, a very similar potential for self-similarity and static fields exists, but these comparisons

are made *between* generations of the algorithm, rather than along the length of the string.

Consider the following L-system rule; in the case below, the axiom is 80 symbols long

(39 symbols of 'W', one 'B' symbol, then 40 more symbols of 'W'):

\square :	99*W, B, 100*W
p1:	BB -> W
p2:	BW-> W
p3:	B<W>B -> W
p4:	B<W>W -> B
p5:	WB-> W
p6:	WW-> W
p7:	W<W>B -> B
p8:	W<W>W -> W

Figure 3.17: a simple CA defined as a context-sensitive L-system

Note that the above Lindenmayer system contains no wildcards in the context fields. All eight possible combinations of a cell and its two neighbors are covered explicitly by the production rules.

The most common way to visualize this L-system is to view each successive generation as a row of white and black tiles progressing from top to bottom in a grid. The string represents the row, so we begin with 200 tiles, all of which are white ('W') except for one black tile ('B') in the center. Depending on the color of each tile, as well as the color of its neighbors, the tile on the row below is either black ('B') or white ('W'). The first 100 generations of this L-system look like this:

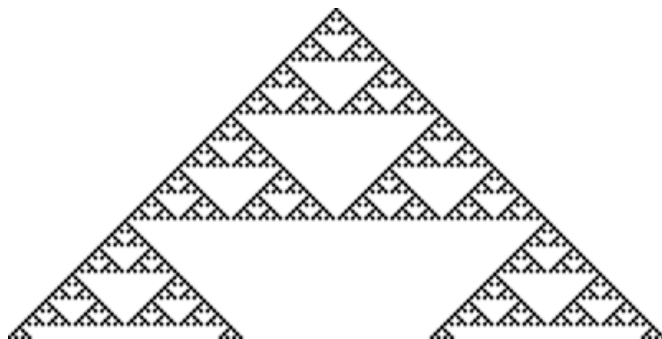


Figure 3.18: the first 100 generations of the L-system in Figure 3.17

This L-system traces the shape of a well-defined fractal pattern known as the Sierpinski triangle (Sierpinski, 1916). It features obvious characteristics of organizational self-similarity, as well as a continuous series of mirrored bifurcations in the pattern (creating the illusion of lines splitting apart and moving away from one another in contrary motion).

If we realize this pattern as music with each generation mapped to a musical event, we can see some of the patterns emerge. We could use the letter 'B' to represent a note at a particular pitch, and map the center 80 cells of the axiom to a pitch space running from, say, G#0 to E7 (MIDI note numbers 20-100), for the first 128 generations. The music from such a mapping (with each generation occupying an eighth note of musical time) appears on the next page (Figure 3.19).

We can see from the music we've generated how the fractal bifurcation of successive generations of the string translates into polyphonic density. What begins as a monophonic line quickly grows to a density of 16 pitches by the sixteenth beat. It then collapses back to a two-note line, increasing again to 16 and then 27 notes on the 24th and 32nd beat. This increasing density is reflected in the shape above, as is the distance between polyphonic lines and clusters (which increase over every growth-collapse cycle) and the contrary motion of the lines (explicit in the entire pattern, but most obviously traced in the outer voices of the clusters).

By treating a Lindenmayer string as a musical singularity (even if we still unwrap it in time), we can easily investigate musically interesting patterns that occur between generations of the string. However, the deterministic mapping we've been using has its

limitations. By treating some members of our Lindenmayer alphabet as metadata, however, we can achieve some very interesting results.



Figure 3.19: the first 128 generations of the L-system in Figure 3.17

Parametric symbolic mapping (metadata)

The mapping strategies we've considered thus far have dealt with an L-system string as a directly evaluated stream of events, evaluated in series or simultaneously.

Symbols in our Lindenmayer alphabets are unlinked except in the rather trivial sense that

they follow one another either in time, pitch height, or another mapping scheme we might devise to generate a stream of musical information. In this section we'll introduce some examples of musical parsers to interpret symbols as musical *metadata*, i.e. instructions that influence subsequent musical events. Rather than being mapped to events themselves, these symbols can be mapped to instructions to a virtual musical interpreter akin in scope to a turtle interpreter.

If we revisit our turtle graphics instruction set for a moment, we remember that in most Lindenmayer parsers, symbols are not treated equally in terms of their direct consequence. Often, the true meaning of a symbol only becomes clear when one considers its *context*. Symbols earlier in the string influence symbols later in the string, and a symbol at any given point is defined only by the symbols leading up to it, even though the grammar model uses the same systems and rules for generating all the symbols in the alphabet. To put an Orwellian spin on it, all of our symbols are equal (to the L-system generator), but some symbols are more equal than others (to the interpreters we devise).

For example, if we look at the following L-system:

□:	F		
p1:	F	->	+GQ
p2:	G	->	FY-F
p3:	Q	->	-FG+
p4:	Y	->	+GF-

Figure 3.20: an L-system with a six-symbol alphabet

The raw L-system strings for generations 0-5 of this system look like this:

Gen 0: F
 Gen 1: +GQ
 Gen 2: +FY-F-FG+
 Gen 3: ++GQ+GF--+GQ-+GQFY-F+
 Gen 4: ++FY-F-FG++FY-F+GQ--+FY-F-FG++FY-F-FG++GQ+GF--+GQ+

Gen 5: +++GQ+GF--+GQ-+GQFY-F+++GQ+GF--+GQ+FY-F-FG+---++GQ+GF--+GQ-+GQFY-F-
 ++GQ+GF--+GQ-+GQFY-F++FY-F-FG++FY-F+GQ--+FY-F-FG++

Figure 3.21: generations 0-5 of the L-system in Figure 3.20

Suppose this time, instead of a direct mapping of symbol-to-event, we were to map as follows:

F	Sound a note at the current pitch for the current duration.
G	Sound a note a minor third higher than the current pitch for the current duration.
-	Transpose the current pitch down a perfect 5 th .
+	Transpose the current pitch up a perfect 5 th .
Q	Decrease the current duration by 125 ms.
Y	Increase the current duration by 125 ms.

Figure 3.22: a simple instruction set for a musical parser

Let's define the terms "current pitch" and "current duration" to begin as middle C and 500ms, respectively. For the moment, we'll continue to scan the string from left-to-right using a constant timebase of 125ms per symbol (i.e. at a constant tempo of 120 quarter-note beats per minute, with each symbol representing a sixteenth note). As a result, symbols that don't generate notes will advance the time, creating a rhythm based on when the 'active' symbols ('F' and 'G') fall. Generations 0-5 (separated by double bar lines) might look like the music on the next page (Figure 3.23).

By using an instruction set for our Lindenmayer parser, we've introduced elements of *parametric mapping* into our system. Parametric mapping adds a great deal of flexibility to how we interpret our L-system data. By manipulating our instruction set and mode of transversal (the system by which we go through the string) we can generate vastly different musical output for any given Lindenmayer string.

The musical score is organized into five systems, each containing multiple staves. The first system (measures 1-4) shows the initial realization. The second system (measures 5-7) continues the development. The third system (measures 8-10) includes a measure with a fermata. The fourth system (measures 11-13) features a measure with a fermata. The fifth system (measures 14-16) concludes the piece. The notation includes various rhythmic values, accidentals, and dynamic markings.

Figure 3.23: generations 0-5 realized using the instruction set in Figure 3.22

By inserting a flexible parametric parser between our L-system and our musical output, however, we've massively expanded the potential complexity of our mapping system. The risk inherent in this endeavor is obvious; as the ratio of metadata to event data increases, the less our musical output will directly reflect the Lindenmayer input.

With this in mind, the important concern of how to maintain musical salience in terms of the features of the L-system in question needs to be addressed. While there exists no sure formula for designing a parser system, a few ideas come to mind.

Generally speaking, the features of the Lindenmayer topology that need to be retained in our musical output are those of growth and self-similarity.⁶ These features are expressed not through the alphabet of any given system *per se*, but rather through how we apply our production rules to the system. From our exposure to L-systems as a modeling language, we can discern that the symbols that occur as predecessors in the production rules tend to be either 'active' symbols in the alphabet (e.g. 'F' in our turtle parser) or symbols that have no effect on the parser (e.g. 'X' in the fern example, which exists solely as part of the production rules and is ignored by the parser). Symbols which are interpreted by the parser as parametric data ('[', '+', etc.) are seldom included as the main component of a production rule, with one important exception noted below. As a correlate, the primary 'active' symbols in the turtle parser are seldom ignored by the production rules entirely, though there are cases where this happens.

⁶ This is assuming, of course, that we want our musical output to in some way *sound* like it could have come from an L-system. If we're simply "mining" our data to generate interesting numerical material that we can translate into music, the task of making the self-similar metaphor musically salient becomes much less important. The ideas presented in this paper are all somewhat geared to the author's particular goals as a composer, i.e. to generate algorithmic music that explores the features of the algorithm used in a vivid and interesting way.

We can restrict our instruction set based on this observation to allow as ‘active’ symbols only those parts of the alphabet that are integral to the production rules. In addition, we could map parametric commands only to those symbols that are not matched as predecessors in our production rules. The alphabetic subset used in the production rules would then be comprised of only those symbols that are ‘active’ to the parser or important to the L-system we want to create. Parametric symbols would only be included in successor symbols or context matching.

One exception to the guidelines above, however, would permit the inclusion of parametric symbols into the production rules. This exception occurs when part of the L-system relies on *complementary* symbols. In our turtle parser, these would include the ‘-’ and ‘+’ symbols, which are inversions of one another. In our initial musical parser above, the symbols {-,+} and {Q,Y} are complements of each other, and will often feature in production rules where they alternate one another, e.g. Q->Y and Y->Q. This is common in L-systems where the polarity of some of the fractal features alternate on successive generations.

The L-system in Figure 3.20 could be re-mapped using a different (and in some ways simpler) instruction set as follows:

F	Increment the current pitch by 3 semitones then play a note.
G	Decrement the current pitch by 4 semitones then play a note.
Q	Increment the current pitch by 5 semitones then play a note.
Y	Decrement the current pitch by 7 semitones then play a note.
+	Speed up note delta time one metric value.
-	Slow down note delta time one metric value.

Figure 3.24: another instruction set for a musical parser

With this parser, we have changed a number of things about how we interpret our string. Firstly, we’ve decoupled our timebase from the pacing of the string. Put more

simply, our parser now controls the amount of musical time between events, depending on whether events occur and on the current state of a variable (the “note delta time”, afterwards NDT). As a result, time only advances when ‘active’ symbols are read from the string, though the actual distance between events is determined by parametric information. The mapping table for the note delta times in the musical example that follows is also a bit more sophisticated than simply adding and subtracting to the duration of notes, as we were doing previously. For this parser, we decide on a 4/4 metric grid. NDT values in the range below four beats (an entire measure) change by increments of *exponential* values. For example, if our current NDT is a half note, a symbol ‘+’ will increment the pacing to a quarter note. Further ‘+’ symbols will speed up the NDT to eighth notes, sixteenth notes, thirty-second notes, up to a salience threshold we can define ahead of time. NDT values above four beats in length, however, change in *linear* increments of quarter notes, so that successive ‘-’ symbols will change an NDT of four beats to five beats, six beats, seven beats, etc.

Secondly, we’ve made the four symbols featured in the production list ‘active’ as well as parametric. The symbols ‘F’, ‘G’, ‘Q’, and ‘Y’ all generate a note in the music, but they determine their pitch by making a relative intervallic jump from the previous note’s position prior to sounding. By placing the transposition before the actual note, we perform the musical equivalent of a prefix rather than a postfix operation on the current pitch height. As a result, the symbol ‘F’ will always sound a minor third above the previous note. If we had done this the other way around (sounding the note, then transposing), the pitch sounded by an ‘F’ would depend entirely on the transposition level set by the previous ‘active’ symbol, which would miss the point.

The following example uses generation 5 of our L-system (see Figure 3.19). We have set our initial pitch to C 5 and our initial note delta time to a half note. With some manual intervention in terms of note durations and phrase markings, we could get something like this:

The image shows a musical score for a piece titled "curious...". It consists of six staves of music, each starting with a measure number: 4, 8, 11, 14, 17, and 20. The music is written in treble clef with a key signature of one sharp (F#). The tempo is marked as 140. The score includes various musical notations such as notes, rests, and dynamic markings like "rit." (ritardando) at the beginning of the sixth staff. The music is characterized by complex rhythmic patterns and intervallic leaps.

Figure 3.25: a musical realization of generation 5 of our L-system, using the instruction set in Figure 3.24

If we compare this parser output to what our previous parser yielded, we could definitely argue that our new instruction set is a big improvement. There are a number of reasons for this, not least being that we've made sensible choices in which symbols are mapped to 'active' musical events, versus which symbols we relegate to behind-the-scenes operations on our musical variables. By binding intervallic leaps and note events together, we've created a framework where every sounding note has a pitch derived from

its value as a Lindenmayer symbol, matched against the previous pitch. In addition, we've chosen unique intervallic leaps for our symbols (even though 'Q' and 'Y' are similar in that they both transpose through a perfect 4th cycle, albeit from opposite directions). As a result, we've created a *deterministic* symbol mapping. This is why repeated patterns in our Lindenmayer string in this mapping yield motivic fragments with common interval relationships (e.g. 'FYFFG' will always yield some transposition of the interval vector {3, -7, 3, 3, -4}).⁷

Before we get carried away, however, we should look at a few other reasons why this particular mapping seems to work much better than its predecessor in Figure 3.20. One possible reason is simply that we've implicitly designed our mapping towards maintaining a monophonic texture. While there are many mapping strategies which can generate independent polyphony from a single string, mapping production symbols to note duration is probably not the way to do it. Our preferred method, which we'll look at in the next chapter as we investigate real-time scenarios, is to use *branching*.

Another reason why this mapping seems to adequately reflect some of the characteristics of the L-system string has to do with our choices in mapping timing and pitch. By using an exponential scale for short NDT values and a linear scale for long ones, we evade the pitfall of creating incredibly long gaps in the score (which would happen if we stayed in the exponential realm for long durations); similarly, a linear mapping would have caused absurdly complex NDT values in the shorter range.

⁷ Because of the deterministic quality of our mapping strategy, we could decode this piece of music backwards as a ciphertext, and generate all the 'active' symbols of the Lindenmayer string we started with!

As for pitch, we were fortunate enough to pick a mapping that kept the melodic line within a reasonable boundary range, creating a much more satisfying experience than, say, the way we mapped generation 5 using the earlier mapping. One of the ways we accomplished this was to pick interval relationships that reflected the statistical frequency of the equivalent symbols in the string. Let's look at how we came to that decision.

Generation 5 of our L-system has 117 symbols in the string. A histogram of the six symbols in our alphabet yields something like this:

Symbol	# of times	% of total string
F	21	17.948717%
G	19	16.239316%
Q	12	10.256410%
Y	7	05.982906%
+	32	27.350427%
-	26	22.222222%
Total	117	~100%

Figure 3.26: statistical breakdown of the symbol frequencies in our L-system

We can find out a few things about how our musical mapping will play out just by looking at these numbers. For example, the fact that there are roughly twenty percent more '+' symbols than '-' symbols in our string indicates to us that if we bind a complementary mapping to those two symbols, the '+' will gradually dominate the mapping. For example, if we had mapped those two symbols to equally measured increases and decreases in volume, respectively, the end of our musical phrase would be significantly louder than the beginning. In other words, despite local incidences of crescendi and decrescendi, our piece overall would crescendo. Obviously, we could view this as a feature of this particular Lindenmayer string and enjoy that artifact of the mapping as generating an overall arc to the piece that might otherwise be lacking.

In our present case an anomaly occurs. If our statistics are true (and we're reasonably sure they are), then our piece should get faster as it goes along, and not (as it seems to) slow down towards the end. However, we're forgetting to take into account the *placement* of these complementary symbols along the string. Three of our '+' symbols occur at the very beginning, effecting not a change in pacing, but simply changing the NDT of the first note to a sixteenth note from its initial value as a half note. Two of our '+' symbols occur at the end of the string, where they do nothing. As a result, our score is only 27 to 26, hardly a difference. The slowdown is further accentuated by the fact that the density of '+' symbols and note-producing symbols is slightly skewed towards the beginning of the string, resulting in a sparser texture as we progress through the interpretation.

Looking at the histogram of our 'active' symbols ('F', 'G', 'Q', and 'Y'), we see that each is slightly more frequent than the next as we go down the alphabet, respectively. Just as with the statistics for our '+' and '-' symbols, this is a byproduct of the production rules, which may substitute one symbol more frequently than others. Rather than making the symbols directly invertible in pairs (as we've done with '+' and '-'), we've decided to pick interval values that more-or-less offset one another, i.e.:

$$72 \text{ (our starting pitch)} + 21*3 \text{ ('F')} + 19*-4 \text{ ('G')} + 12*5 \text{ ('Q')} + 7*-7 \text{ ('Y')} = 70$$

As a result, we only drift two semitones over the course of the entire piece (if we start from the C 5 we never hear).⁸

⁸ Statistical analysis can be a vital tool to any algorithmic pre-compositional process, whether the data sets are deterministic or not. Whenever we insert an algorithm that changes a musical parameter using *relative* steps, we run the risk of the data causing the parameter to jump off the scales. Now that the 'compute time' of many of these algorithms has decreased to virtually nothing (i.e. we can hear the results as soon as we

We've seen in this chapter how mapping is the crucial force in determining how to apply Lindenmayer systems towards the generation of musical information. In the next chapter, we'll explore the use of L-systems to generate musical accompaniment to a real-time axiom (i.e. a performer) as well as look at some ways to use branching to generate a more complex texture. Finally, we'll discuss some of the ways in which we could integrate L-systems into real-time performance systems to influence not only the musical narrative, but acoustic phenomena as well.

make the algorithm), it's easier to correct for these discrepancies as we work, but it's often worth our time to compute the median transform (as well as the boundaries of our transformation) ahead of time, to make sure our parameter remains in a manageable range. These same histograms (and other statistical methodologies) are equally useful in comparative analyses of existing pieces of music, and are widely used in music research (see e.g. Foxley 1982, Temperley 2001).

4. Interactive Performance Using L-systems

In this chapter we'll take a look at some of the ways to implement Lindenmayer systems in *real-time* as a way to generate musically interesting output from a live performance. In the last chapter, we investigated some of the various mapping schemes for generating musical material from Lindenmayer strings. What follows are some thoughts on how to do the reverse; we'll be taking musical material and looking at how to transform it into data that can drive an L-system, thus generating a string that can be interpreted to create a musical transformation of or accompaniment to the incoming musical stream. We'll close up our investigation with some ways in which these new methodologies (as well as our mapping schemes from the previous chapter) can be integrated into a functional interactive performance system that runs reasonably well when driven by a human performer in real-time.

For the purposes of this chapter's discussion, we'll first have to explain what we mean by a *real-time* performance system. The scope of that expression ('real-time', and its slightly more fashionable cousin, 'interactive') is very broad, and is often used in a misleading way. The real-time system outlined in this paper fulfills the following somewhat restrictive definition of a real-time performance system.

Our *interactive real-time music performance system* is an environment wherein a listening agent (in this case a computer) can interpret and process musical or acoustic data from a live performer as soon as it is received, and generate some sort of process based on that data, without having to know *explicitly* the musical information that will be received and when it will receive it. Furthermore, the system must be designed in such a way that it functions in a reasonable manner regardless of the musical actions of the

performer driving the system, accomplishing this task through explicit data reduction, boundary conditions, and other selective filters on the performer input.

Our system (outlined below as it was in Chapter 1) consists of a number of modules. The bulk of our discussion below will focus on the ‘Lindenmayer interpreter’ (and to some extent the ‘symbolic pre-processor’ and ‘musical interpreter’). However, a walkthrough of the rest of the system is appropriate to see how it will fit into the grand scheme of things in our interactive system.

Overview of *Scheherazade*

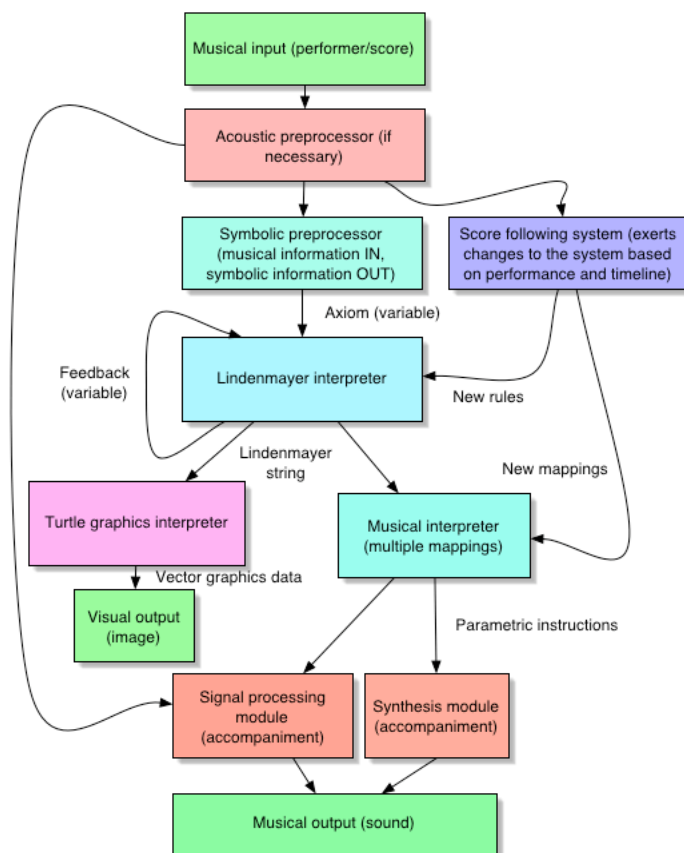


Figure 4.1: Flowchart of the *Scheherazade* system for real-time performance

Our system is based on the premise that the main stream of control data comes from acoustical information provided by a live performer. The audio stream from the live performer is digitally captured by the system and passed to the ‘acoustic preprocessor’.

The ‘acoustic preprocessor’ has the task of taking the sound from the performer and doing two things with it:

- The module cleans up the audio signal so that it can be used as source material for the ‘signal processing module’. This can be a reasonably simple set of procedures (such as equalization or level limiting) or a more complex undertaking, including noise reduction, stereo imaging, etc.
- The module performs ‘data mining’ on the audio signal to generate musical control data for the ‘symbolic preprocessor’ and the ‘score follower’. This involves such things as determining the fundamental pitch of the performer (“pitch-tracking”), describing an amplitude curve of the performance dynamics (“envelope-following”), and parsing the acoustic stream into discrete musical events (“attack detection”) to limit the data passed on down the module chain to only those events which are relevant for the other modules. These tasks are performed by so-called ‘listening agents’ (Rowe, 1993) that act independently of the rest of the system, providing it with information on the ‘current state’ of the performance. If necessary, a hardware device can accomplish much of this work, relieving the computer of the substantial overhead of parsing musical information out of the acoustic stream.

The 'symbolic preprocessor' takes the musical information provided by the 'acoustic preprocessor' and converts that information into a real-time stream of axioms for the generation of Lindenmayer strings. How we do this and what we hope to accomplish with this module will be discussed in depth below.

The 'score following system' listens to the musical data and uses it to track a performer's progress through a piece of music. As the performer plays through the piece, this module triggers appropriate changes to the behavior of other modules in the system. The score follower needs to be designed in such a way as to be able to flexibly ascertain where a performer is in particular piece. It does this by having a copy of the performer's part stored in memory, comparing what the performer actually does with what they should be doing at a given time. In more open-form scenarios (such as improvisatory pieces), the score follower can consist of a 'cue list', allowing a performer to play over a certain section of the piece for an arbitrary length of time until a predefined 'cue' is performed (such as a specific sequence of notes). The follower then makes changes to the system, and waits for the next 'cue'.

This is by no means the only way to follow through a piece, however. Developers of interactive music systems often implement systems that process performer input based on explicit time-based instructions. While these systems are still interactive in the sense that the performer can respond musically to the computer, some of the flexibility of the system is jettisoned in favor of building a system that works deterministically, regardless of what the performer does. These systems are functionally equivalent to playing along with a sequenced series of events; even if those events may depend at the microcosmic

level on data acquired by the performer, at the macrocosmic level they are still running along to a master timeline over which the performer has little or no control.

At the other extreme, however, are systems that require direct tactile intervention to progress through changes in a piece. These systems require some sort of command issued by something outside the musical stream (e.g. a mouse click, a MIDI pedal) to change their behavior, and have a number of drawbacks as well, particularly when utilized by non-specialists.

The use of a score follower allows a composer to have a certain amount of flexibility in terms of the pacing of the performance without having to rely on an additional level of control interface for the computer program. Our score follower, which allows for as much (or as little) performance flexibility as we need, is based on the EXPLODE follower developed by Miller Puckette for Max/FTS at IRCAM (Puckette, 1990) and updated by David Zicarelli at Cycling'74 as “detonate” (Zicarelli, 1995).

Because the system we've designed has no deterministic clock to it, our computer has no way of knowing for certain what the performer will do next, even if it has a score to following along with and can expect certain results. As a result, most of the investigations into L-systems and live input we'll make in this chapter will work under the assumption that we only know what the performer is playing *now*, as well as over a limited time in the past.

The score follower can make formal changes to two modules that we'll discuss in depth below. The 'Lindenmayer interpreter', which also receives axioms via the 'symbolic preprocessor', outputs Lindenmayer strings based on production rules optimized for real-time input. Those strings are then fed into a 'musical interpreter',

which generates information in a manner similar to that described in the previous chapter, but with a framework added to make the output relevant for real-time musical accompaniment. Both of these modules can change their behavior in response to structural instructions issued by the score follower. In addition, a turtle interpreter can also be driven by the same Lindenmayer string as the ‘musical interpreter’, provided the two systems share a compatible alphabet. This allows for synchronized generation of music and graphics.

The ‘signal processing module’ permits the transformation of the performer’s *sound* by the computer, in response to the commands passed down from the ‘musical interpreter’ in response to the performer. The exact way in which this module behaves internally is completely flexible,¹ but it is designed in such a way that different interchangeable modules can be used for different pieces of music, or even within the same piece. A set of music synthesis algorithms (in the ‘synthesis module’) can also be driven by the ‘musical interpreter’ to generate accompaniment for the performer. Either of these two modules can be offloaded to hardware synthesizers and signal processors, provided some mechanism is in place to get data from the ‘musical interpreter’ into a form they can easily understand (typically MIDI).

Now that the system has been outlined somewhat, we’d like to look in depth at the three modules that concern us in determining how to translate musical data into information that can drive an L-system that generates musical accompaniment. The

¹ That is to say, any signal processing routine that takes an acoustic signal and transforms it according to some parametric information (provided by the ‘musical interpreter’) can be used in this module. For example, an echo unit where the rhythmic spacing of the different echoes is defined by the L-systems would be a perfectly good candidate for this module.

‘symbolic preprocessor,’ which needs to translate musical information into symbols compatible with the L-system grammar, the ‘Lindenmayer interpreter,’ which runs the L-system routines on the information output by the ‘symbolic preprocessor,’ and the ‘musical interpreter,’ which takes complete Lindenmayer strings from the ‘Lindenmayer interpreter’ and translates them back into musical information.

Symbolic encoding of musical information

One of the ways in which we can use Lindenmayer systems in real-time is to encode a stream of musical information as a set of symbols that can serve as an axiom for an L-system. How we encode music into these symbols, and whether the encoding is on single events or over a larger time period is another issue of mapping which we will have to resolve. First we have to classify the different types of information we can receive from our acoustic preprocessor.

Let us assume that the system we’ve designed can determine three types of information about musical events input into the computer by a musical performer: pitch, amplitude, and duration (encoded as note delta times between successive events). In order to make the process of following the performance (“tracking”) more fluid, we can limit the scope of the information by filtering out events based on certain criteria. For example:

- We could reject notes below a certain amplitude (which may be mistakes).
- We could reject notes that are known to be outside the musical range of our performer’s instrument (which *must* be mistakes made by the pitch detection algorithm).

- We could reject notes that fall below a certain NDT from the previous event and are of the same pitch (these notes are probably erroneous double attacks).
- We could limit the data flow from notes that change over time without re-attacking (e.g. glissandi) by only listening to the beginning and final pitches of the note.

For our purposes, we'll also assume that the performer input can be encoded as a series of *monophonic* streams (e.g. if a violin performer plays a double stop, we can access the two pitches as separate entities when we encode them as symbols). This is not always a trivial exercise, and does not imply that any perceptual sense of independent lines will be maintained. As a result, we can assume that we'll get all the pitches played, but we might not be able to immediately ascertain which pitch came from which previous pitch if we have a succession of diads or triads. This would involve a regression analysis that may slow down the system.²

For each event played by our performer, therefore, we'll get three numbers from the preprocessor (a pitch, an amplitude, and an NDT value), as well as some possible metadata (e.g. a flag could be set to tell us that the latest pitch is not a new event, but is a glide from a previous pitch).

A list of the first six notes of 'The Star Spangled Banner', encoded as triplets of numbers, might look something like this:

² See Rowe, 2002 for how one might do this if it was necessary.

Pitch	Amplitude	NDT
67	100	0
64	97	300
60	105	100
64	101	400
67	89	400
72	110	400

Figure 4.2: ‘O Say, Can You See!’ encoded as a sequence

The musical data is encoded using MIDI pitch and velocity values for the first two columns (in MIDI, data is usually in the range of 0-127). For pitch, MIDI maps middle C to value 60, with semitones above and below that pitch changing the value in increments of 1 (e.g. the first note of our sequence is the G above middle C, or the pitch 7 semitones above 60). Amplitude is mapped in a similar range and is referred to in MIDI parlance as velocity (reflecting the bias of MIDI towards keyboard performance). The NDT values are the number of milliseconds elapsed since the last note (the start of the performance has a NDT of 0, since it’s the first note we hear).

We can create a Lindenmayer preprocessor that ties symbols to any of these events, or all of them, or treats them as a unit. If we take the simplest possible mapping, we could deal at first with the pitch and encode it into a Lindenmayer alphabet. By performing a modulo-12 operation on the pitch stream (preserving the octave as a separate list of data by dividing the raw pitch by 12), our sequence would look like this:

PC	Octave
7	5
4	5
0	5
4	5
7	5
0	6

Figure 4.3: ‘O Say, Can You See!’, pitch classes and octaves only

The ‘PC’ column of our field is virtually a single-symbol alphabet already. If we take a hint from set theory and encode pc 10 as ‘T’ and 11 as ‘E’, we can use the pitch classes directly in an L-system.

L-systems as transfer functions

A simple L-system could encode each possible pc as symbols in our alphabet.

We could then devise a deterministic, context-free L-system that looks like this:

\square :	PC_n		
p1:	0	->	7
p2:	1	->	8
p3:	2	->	9
p4:	3	->	T
p5:	4	->	E
p6:	5	->	0
p7:	6	->	1
p8:	7	->	2
p9:	8	->	3
p10:	9	->	4
p11:	T	->	5
p12:	E	->	6

Figure 4.4: a simple symbolic transfer function

This L-system is a simple example of a *transfer function*. In our system, every pitch class (denoted as PC_n where n is the current time) is mapped to another pitch class seven semitones higher. This effects a transposition of our input stream up a perfect 5th (or down a perfect 4th). Our musical sequence would then play ‘The Star Spangled Banner’ in G major. By mapping all the symbols in this way, our transfer function is key agnostic, so that music fed in with a different key will still be transposed correctly. Note that this system only encodes the pc symbols, and not the octave of the sounded note, so some notes may be transposed into an incorrect octave when they are rejoined with the octave information from the preprocessor. To correct this, we would need to expand our

alphabet to include more than one octave of symbols to indicate transposition outside of the 0-E alphabet.

It's also worth noting that our transfer function will work well using the recursive metaphor of the L-system. Successive generations of the L-system on a single-note axiom will transpose the incoming musical stream around the cycle of 5^{ths}. Generation 5 of our L-system, then, will transpose our C major melody into B major.

We could map our function to do arbitrary pitch remapping. For example:

\square :	PC_n		
p1:	0	->	0
p2:	1	->	0
p3:	2	->	8
p4:	3	->	3
p5:	4	->	7
p6:	5	->	5
p7:	6	->	3
p8:	7	->	3
p9:	8	->	T
p10:	9	->	2
p11:	T	->	T
p12:	E	->	5

Figure 4.5: a more complex symbolic transfer function

This mapping scheme translates multiple pc inputs into the same pc output (e.g. pitch class 3, 6, and 7 all yield 3 on the output). If we show generations 0-5 of a chromatic scale played through this pitch mapping one after another, we get the following:



Figure 4.6: chromatic scale played through our transfer function (generations 0-5)

Note that by generation 3 there is no change from generation to generation. By mapping more than one input pitch to the same output pitch, our transfer function has created a number of *attractors* in the system, whereby notes at any pitch will eventually gravitate towards pitch classes 0, 3, 4, and 10.

By performing a transfer-based L-system on a musical input stream, we can generate a single accompaniment line to go along with the performed input. As the original sound of the performer will (typically) be heard along with the output from the performance system, this simple mapping is most useful to add deterministic harmony lines to the system.

L-systems as symbolic filters

A slightly more involved mapping would involve treating the pitches acquired by the preprocessor as a series of interval vectors. Thus, instead of {67, 64, 60, 64, 67, 70} as our musical input for the national anthem, we'd be working on a remapping of the numbers {0, -3, -4, 4, 3, 5}. Rather than designing an alphabet around all possible unique chromatic intervals for the given input, we could assign an alphabet based around a three-symbol axiom for each interval. Our L-system could use 'H' and 'L' to represent intervallic direction (higher or lower), our serial numbering (0-9 plus 'T' and 'E') for the specific intervals, and symbols to denote leaps of more than an octave ('O' for more than one octave in either direction). The third symbol in the sequence could be omitted or replaced with a dummy symbol (e.g. '.') if the interval in question is within one octave. For example, if we take the following melodic fragment:



Figure 4.7: melodic fragment

A list of interval vectors for this melody would be $\{0, -4, -2, -5, 14, 1, -7, -3, 8, -7, 1, -1, -3, 12, -7, -1, -1, -2, 9, -10\}$. Our Lindenmayer pre-processor would encode these intervals as:

H0. L4. L2. L5. H2O H2. L7. L3. H8. L7. H1. L1. L3. H0O L7. L1. L1. L2. H9. LT.

We could develop an L-system to remap just a few of these intervals to create a divergent accompaniment line, e.g.:

\square :	$PC_n - PC_{n-1}$		
p1:	3	->	0
p2:	4	->	7
p3:	8	->	2
p4:	2	->	T
p5:	1	->	3

Figure 4.8: a more selective symbolic transfer function

We could apply these re-mapped intervals to the incoming pitch material in two possible ways. If we start with the accompaniment on the same initial pitch as the input and start transforming our material from there, we could remap the intervals based on the previous *output* pitch, so that the remapping has a cumulative effect. Alternately, we could remap the intervals based on the previous *input* pitch. The L-system then works according to the metaphor of a *filter*, with the two algorithms described in the previous sentence corresponding to a recursive filter (output is derived from previous output) or a

non-recursive filter (output is derived from previous input). The two output melodies are shown below (on separate staves below the original).

Figure 4.9: the melody in Figure 4.7 transformed by the L-system in Figure 4.8 (original, recursive transformation, non-recursive transformation)

To explore the two methodologies a bit further, we can look at some differences in how the two versions of our accompaniment line play out. Both of them start on the same pitch (C#) and move to the same note as the first interval (F#, remapped from A because of the 7->4 production rule). The next interval in the list is -2, which becomes remapped to -10 in our production rules. The recursive application of the L-system applies that -10 interval to the previous value *output* from the L-system rather than *input* from the performer. Therefore, we generate a note 10 semitones below the F#, which gives us a G# below middle C. The non-recursive version of our melody applies that -10 interval from the last pitch *input* into the system, i.e. A (the second note of our original melody). As a result, we get the B below middle C as our accompaniment note. A vivid

example of how these approaches vary is in the second beat of measure 3, where the sixteenth-note pattern tied over from the first beat (E, A, Bb, A) gets remapped quite differently. It's worth noting that the recursive filter maintains intervallic relationships internal to its own accompaniment line, whereas the non-recursive filter constantly 're-synchronizes' its starting pitch to the original melody, so that intervallic consistency is sometimes lost. However, the non-recursive implementation is *band-limited*, in the sense that our non-recursive accompaniment line will never stray too far in terms of range from the original. The recursive algorithm, on the other hand, can easily veer through successive large output intervals well out of the melodic range of the input melody.

Although it might have escaped notice, our Lindenmayer string derives its running axiom in real time by computing the interval vector as the *difference* between the latest performed pitch class and its immediate predecessor ($PC_n - PC_{n-1}$). We could create a musical accompaniment in a similar way by generating production rules based on the sum of the current and last pitch classes (an aggregate), or the average, for example. By adding more and more past pitches into the computation we can sculpt an accompaniment line based on more musical material, allowing for a more subtle application of the process. This is analogous to varying the order of a filter in signal processing, allowing for a more contoured alteration of a sound spectrum. As with higher-order filters, the amount of delay before the system starts to function appropriately will grow as we look back further and further in time for information. If we are at the beginning of a performance and we need to look back four events to construct our L-system axiom, we won't have anything to drive the L-system until the fourth note of the piece.

Figurative encoding

Another way in which we can encode the real-time musical stream is to simply use L-system symbols as *placeholders* for any musical data. By doing this, we assign a symbol in our L-system alphabet to represent the current musical event (whatever it is), and then generate a system to create an accompaniment accordingly.

For example, if we take our L-system from the previous chapter:

□:	F		
p1:	F	->	+GQ
p2:	G	->	FY-F
p3:	Q	->	-FG+
p4:	Y	->	+GF-

Figure 4.10: an L-system with a six-symbol alphabet, taken from chapter 3

If we bind the symbol ‘F’ to represent the current pitch, we can create an accompaniment line where every pitch is run through the L-system equally, spawning an independent melody off of each note. To do this, we’ll use generation 3 of this L-system ($++GQ+GF--+GQ+GQFY-F+$), and apply it to every incoming pitch. We’ll also use the same L-system musical parser we developed towards the end of last chapter, i.e.:

F	Increment the current pitch by 3 semitones then play a note.
G	Decrement the current pitch by 4 semitones then play a note.
Q	Increment the current pitch by 5 semitones then play a note.
Y	Decrement the current pitch by 7 semitones then play a note.
+	Speed up note delta time one metric value.
-	Slow down note delta time one metric value.

Figure 4.11: our instruction set for the L-system in Figure 4.10

The way we approached this particular L-system (and parser) for pre-compositional scenarios is to assign a starting pitch and duration to feed into the L-system. To use this as a real-time engine, however, we have a decision to make. If we want the accompaniment to start immediately (so that when the performer plays a note

the L-system spawns an accompaniment line), we have no way to assign a starting duration correctly (i.e. we have no idea how long the performer is going to play the current note for). On the other hand, if we wait until the performer finishes the note, and start the accompaniment line when she or he plays the *next* note, we can attempt to derive the duration to feed into our L-system based on the NDT time of the new pitch (which will be greater than or equal to the duration of the previous note). Using the latter system, however, may cause more mistakes than we think, given that we have no way of preventing against the performer playing a note, waiting ten seconds, and then playing another note (causing a huge NDT for our L-system to start with). For our purposes, then, it might be best to rely on a starting duration value derived from some other source, such as the score following module, which may have a general idea of what duration the current note might be.

If we feed in our melody from Figure 4.7 (above) into this L-system, with each note coming in as the axiom of a new run of the grammar model, we could get something like the bottom staff of the line below. We've assumed that all notes start at an equal duration (a half note), and we've removed some of the tied note values for clarity.

Figure 4.12: the melody from Figure 4.7 played through generation 3 of the L-system in 4.10

The primary difference between generating an entire string of musical material with every new input note rather than our previous techniques of doing simple operations on a “one-note in, one-note out” basis, is that the amount of data generated by the L-system is highly amplified in relation to the amount of musical material we feed into the system. As a result, the effect created by ‘spawning’ a new accompaniment line with every note correlates strongly to the impression of a melodically varied echo, with the timing and pitch effects of the echo constructed by the L-system. In our example above, every input note will create eleven accompaniment notes. A single input note at C 5 would create this accompaniment:



Figure 4.13: a single note played through generation 3 of the L-system in 4.10

Using single notes to test the responsiveness of a particular L-system is a good way to determine how they'll fit in with the musical input. However, it should be noted that, as with any delay line, complex input rhythms generate exponentially complex output rhythms as the different accompaniment lines intersect. This holds true even if the rhythm created by the single note 'impulse' to the L-system is reasonably simple.

Branching as polyphonic accompaniment

Two Lindenmayer symbols in common use in turtle interpreters are *branching* symbols. Traditionally, these are indicated as '[' and ']' in L-systems, and indicate the start of a separate graphical child stream, that inherits its initial orientation from the parent but proceeds from it independently. We could incorporate branching structures into our L-system for music by using branches to indicate the generation of independent polyphonic lines, that start at the pitch of parent branch but then move independently away from the parent line.

For example, if we generate a running axiom of a performer's input that encodes the current pitch class (PC_n) as 'F', the previous pitch class (PC_{n-1}) as 'X', and the pitch class before that (PC_{n-2}) as 'Y', we could perform an L-system that would manipulate three pitches simultaneously. We could introduce branching into the accompaniment using the following L-system as an example:

□:	FXY		
p1:	F	->	++FY
p2:	Y	->	-F[X][-Y+F]
p3:	X	->	X[F+Y]

Figure 4.14: an L-system that uses branching and starts with a three-symbol axiom that references the last three notes played by their pitch-classes

Each triplet of symbols is input into the system at once on every three notes of the performance. If we use generation 2 of the L-system as our processing model, we would spawn an accompaniment line that follows the string:

++++FY-F[X][-Y+F]X[F+Y][++FY+-F[X][-Y+F]]-++FY[X[F+Y]][-F[X][-Y+F]]++++FY]

If you look at the string above, you'll notice that the string has bifurcated in a number of places to create branches. If we map out the string so that the branches form additional rows of a string, we'd get something like the image below. The arrows indicate how the different branches spawn relative to their previous symbol. Different colors indicate different branches and sub-branches.

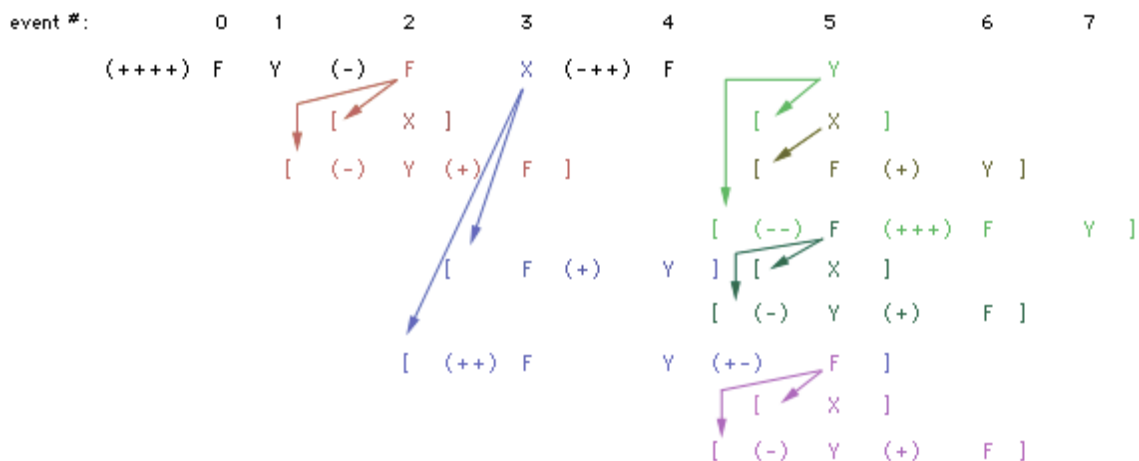


Figure 4.15: the string generated by generation 2 of the L-system in Figure 4.14, mapped out as separate branches for each row

As a new branch spawns, it inherits whatever state the parser was at when we move to the next branch. This means that the ‘X’ on line two performs its actions *relative* to the symbols leading up to it on the previous line (‘F’). However, when we return to line one (with the ‘X’ on event number 3), the system returns to the previous state.

If we treat as ‘active’ symbols ‘F’, ‘X’, and ‘Y’, and leave all the other symbols in our alphabet as metadata, we can treat successive active symbols as events that occur on successive beats. We could start out accompaniment sequence as soon as the third note in the input performance occurs, and interpret the string as follows:

- F Play note PC_n at the current transposition level.
- X Play note PC_{n-1} at the current transposition level.
- Y Play note PC_{n-2} at the current transposition level.
- + Increment the transposition level by 7.
- Decrement the transposition level by 7.

Figure 4.16: our instruction set for the L-system in Figure 4.14

If we interpret every event beat to be a quarter note, and we treat the transpositions to affect only pitch classes (maintaining the correct octave), we could get something like this for a three-note input melody (the performed melody is on the top staff, with the accompaniment below):



Figure 4.17: a musical accompaniment using the L-system in Figure 4.14

Some of the features of this particular Lindenmayer string become salient in the polyphonic accompaniment. For example, the statistical prevalence of ‘+’ symbols over

‘-’ symbols weights the output pitches well into the positive range on the cycle of 5ths. A more balanced L-system would correct for some of that discrepancy. In addition, the nesting patterns of this generation work in such a way that the polyphony varies in an arch throughout the string, reaching a maximum number of active branches three beats before the end of the sequence.

The L-system outlined above is effectively doing a combination of the techniques we’ve already looked at. It computes a running axiom by using the last three input notes (similar to our ‘filter’ using interval vectors). Rather than translating them into a single symbol axiom, however, we’re using all three of them discretely in our L-system. In addition, our L-system interpreter spawns new musical events by offsetting and transposing the notes in the original axiom in a polyphonic context.

Parametric parsing

Now that we’ve looked at some of the ways in which we can use L-systems to generate accompaniment material in real-time, we can ruminate for a paragraph or two on how we would take the data out of the L-systems and use it for something other than simple manipulation of incoming pitches. While for descriptive purposes using pitch transformations is probably the most effective way to demonstrate how the systems spawn accompanying melodic material, there is no particular reason why we couldn’t encode note velocity, duration (NDT values), or any other more parametric data as symbols and have them integrated into our accompaniment strategies.

A further possibility might be to use the incoming pitch information to generate L-system strings that are then interpreted as ‘parametric’ data for synthesis or signal

processing. The polyphonic accompaniment line illustrated above in Figure 4.16, for example, could easily be used to set the resonant frequencies of a filter bank, through which the musical performance could be processed. Similarly, we could use L-systems to generate impulse response material for creating interesting reverberant spaces and convolution processes, allowing the musical performance to sculpt the sound coming out of the computer in a more subtle way.

The parametric data can also influence how the accompaniment is generated. Alphabetic symbols could be added to our L-system to inform the sound-producing modules in the computer to use different samples for the accompaniment line, for example. As has been evident throughout our exploration of these possible configurations, the success of any particular L-system for generating a musical texture depends on the way the data is mapped just as much as what symbolic data the grammar model gives us.

For example, in the piece *Repeat After Me* (see Scores) the final section of the piece uses a delay processing module to alter the sound of the performer (a flautist). Twelve delay units working in parallel process the flute. Each delay unit works at an increasing metric value of a sixteenth note (e.g. delay unit 1 delays the flute by one sixteenth note, delay unit 2 delays the flute by an eighth note, etc.). The delays have variable feedback controls (to create an echoing effect) and have a resonant filter in the chain, which can be tuned to a different frequency. The specific frequencies and feedback amounts are determined by the current event played by the flautist, according to an L-system.

We construct the axiom of the L-system by encoding the pitch class of the currently playing note as a symbol, as well as its volume (encoded as ‘p’ for piano, ‘m’ for mezzo-piano, ‘M’ for mezzo-forte, and ‘F’ for forte). ‘2F’, or example, means that the flautist is playing a loud tone at pitch-class 2 (‘D’). The following L-system is used for each axiom:

\square :	$PC_n A_n$		
p1:	0	->	2
p2:	1	->	3
p3:	2	->	2
p4:	3	->	2
p5:	4	->	6
p6:	5	->	6
p7:	6	->	9
p8:	7	->	0
p9:	8	->	3
p10:	9	->	9
p11:	T	->	9
p12:	E	->	4
p13:	p	->	m
p14:	m	->	F
p15:	M	->	p
p16:	F	->	M

Figure 4.18: the parametric L-system for the echoes in *Repeat After Me*

Each generation of the L-system will transform the pitch classes of the incoming note so that they attract towards pc 2 and 9 (‘D’ and ‘A’) over multiple generations. Additionally, the dynamics of the current note will be shifted (piano input becomes mezzo-piano output, etc.). The pitch-class output by the L-system sets the resonant frequency of one of the delay units. The amplitude output sets the amount of delay regeneration (‘feedback’) in the delay unit. The L-system is run for 12 generations. However, each generation sets the duration of only one unit. Delay unit 1 will be set according to generation 1 of the L-system, delay unit 2 will respond to generation 2, and so forth. The feedback in the delay units guarantees some residual effects; even though

their resonant frequencies and regeneration amounts change with each note, there will still be sound in the echo network from the previous note. In addition, because the echoes are listening to the input signal from as much as three beats ago (12 sixteenth notes), the processing set by the current note is applied to the sound of a note that already happened.

A more direct form of parametric processing occurs softly throughout the piece. An L-system drives a granular processing unit that segments the sound of the flautist into small individual units ('grains'). These grains can be overlapped, transposed, played backwards, etc.

In this processing, the NDT values (the amount of time between notes played by the flute) alter the spacing between grains, according to the transfer function below. The NDT values are encoded as 'S' (short), 'M' (medium), 'L' (long), and 'X' (very long). Symbols output by the L-system set the grain rate accordingly (a grain rate of 'S' means that the grains vary between 10-50ms, 'M'-length grains are in the 50-200ms range, etc.).

□:	NDT _n		
p1:	S	->	L
p2:	L	->	M
p3:	M	->	S
p16:	X	->	M

Figure 4.19: the L-system to set the grain size of the processing in *Repeat After Me*

Multiple generations of the L-system will force the grains to attract towards shorter durations. This accomplishes a simple process that could probably be done in a non-symbolic manner (e.g. using a function table). However, working symbolically allows us to use our alphabet ('S', 'M', 'L', and 'X') to synchronize another process easily. In *Repeat After Me*, we use these symbols in part of the piece to determine the

duration of the samples output by the computer as part of the accompaniment ('S' samples will play very staccato, etc.).

Parametric encoding allows us to use our musical input data to control any variable process in our interactive performance system. The L-systems allow us to add a sophisticated layer of algorithmic processing on the symbolic data, allowing us to create control structures for our system that are less linear and more reflective of the underlying algorithm that we want to explore.

5. Conclusion

We began our discussion in this paper with some ideas on what kinds of algorithmic processes make for good music. While there is no truly perfect answer to this question, what we've presented in the past few chapters are some interesting applications of one particular class of algorithms. Lindenmayer systems are well suited for musical applications for many reasons that we've outlined already. L-systems work according to a grammar model, an underlying scheme of evaluating form that has many applications in music, from how we perceive music to how we go about writing it. As it happens, L-systems also exhibit a high degree of fractal self-similarity, which exhibits itself through the process of database amplification. This allows us to generate extensive musical passages from a very small seed of information, while retaining a degree of coherence throughout the form of the generated output.

The pieces included with this dissertation embody some of the diverse applications for L-systems at all levels of the compositional process. *Growing Pains* (for mandolin or guitar) is a fairly strict musical interpretation of a particular L-system string ("the hypothetical fern"). The computer system generates visual material in synchrony with the music, but the processing and accompaniment are very minimal and only tangentially derived from the L-system itself. *Repeat After Me* (for flute) works according to a somewhat opposite metaphor; the music played by the performer is derived from a composite spectral analysis of some jazz harmonies expanded by an L-system. Much of the figuration, however, is through-composed, to explore some rhythms that work well within the framework of an interactive accompaniment. The processing on the flute, however, is almost entirely derived from different Lindenmayer procedures,

including L-systems that derive a real-time accompaniment, L-systems that determine signal processing on the flute sound, and L-systems that change the timbre of the accompaniment lines. *Biology* is a trio of short pieces that can be arranged for any solo instrument (presented here for violin) where the musical material is derived from morphogenic L-systems. Though the pieces are algorithmically conceived, they are included here as acoustic works with no processing.

As with all algorithmic composition “solutions” we need to resist a number of temptations inherent in how the system works. First and foremost, the algorithmic material generated by the computer will always demand extensive compositional intervention to truly make it into a coherent musical shape. The plants generated by L-systems don’t look like the real plants found in our environment. They merely map out the salient morphogenetic features of specific species of plants without portraying any of the unique characteristics of a particular specimen. In a similar vein, our generated music only works as real music when the particular aesthetic concerns of a composer are applied to the material. While some of this can be done through the algorithmic mapping procedures we’ve discussed, much of the work needs to be done by listening and modifying the musical information, to give it an overall shape and to make the passages generated by the grammar model sound like musical phrases, not just a stream of numbers. While we’d love to say that the computer, the great labor-saving device of the past century, can help us with this part of the task, the truth is that the final, manual intervention of the composer will always be necessary to make the piece really work. In many ways using L-systems to make music requires *more* work in the end on the part of the composer than many other algorithmic procedures. We need to pick the mapping

schemes, the particular L-system, how to configure the real-time input, and how to design the overall form of the piece by varying the algorithm constantly. This may be one of its greatest assets, as our methodology outlined above demands extensive intervention by the composer at all levels of the process. At the end of the day, we may end up with a very rewarding experience. Experimenting with the transcoding of abstract data can give an artist some powerful insights into how different procedures influence the outcome. The end result, however, is what really counts.

Appendix: source code examples

What follows are the source code for three programs written by the author as part of the Cycling'74 Jitter software package for real-time matrix manipulation (Clayton, Jones, Bernstein, DuBois, and Grosse, 2002). The first (*jit.linden.c*) performs Lindenmayer string rewriting on a 1-dimensional Jitter matrix of input symbols. The second (*max.jit.turtle.c*) interprets a Jitter matrix as turtle instructions, generating QuickDraw graphical instructions for 2-D vector graphics that can be interpreted by the Max *lcd* object (Freed, Lee, Ellison, and Zicarelli, 1990). The third (*jit.lindenpoly.c*) interprets a 1-dimensional Lindenmayer system as a polyphonic line, parsing the symbolic information into a 2-dimensional array based on branching symbols.

For more information on Jitter, see the Cycling'74 website (www.cycling74.com). These objects are publicly available as part of the Software Development Kit for Jitter, downloadable from the above website.

The source code examples are all copyright © 2002-2003 Cycling '74 -- All rights reserved. Used by permission.

jit.linden.c

```

/*
    Copyright 2002 - Cycling '74
    R. Luke DuBois luke@music.columbia.edu
*/

// updated for new jit arch by rld -- 2/25/02

/*
 * jit.linden interprets an incoming 1-dim, 1-plane char matrix as a Lindenmayer System (L-system).
 *
 * named after Aristid Lindenmayer (1925-1989), L-systems work on an interpreted grammar model wherein a syntax is defined for
 * replacing individual elements of the incoming string with a replacement string. these models (called production rules, or just
 * 'productions'), can be context-dependent. L-systems can also support multiple modes of branching (only one dimension of
 * branching is supported here, though branches can be nested). generally speaking, L-systems get larger (not smaller) through
 * successive productions (or generations of the grammar); the size of the jit matrix used by the object determines the maximum
 * length of the string, so a large matrix is advisable, even if the axiom (starting string) is very small.
 *
 * jit.linden treats the incoming matrix as ASCII, and certain values are reserved to indicate wildcards ('*' by default) and
 * branching ('[' and ']') by default). these can be set by the 'wildcard', 'leftbranch', and 'rightbranch' attributes.
 *
 * the arguments to the 'ignore' and 'production' attributes should be lists of ASCII characters, e.g.:
 *
 * 'ignore F C A' tells jit.linden to ignore the ASCII characters 'F', 'C', and 'A' when checking context during a production match.
 * typically ignored symbols are characters which describe geo-spatial information (e.g. turtle graphics) and aren't included in the
 * grammar of the production.
 *
 * 'production * F * +F[F]' tells jit.linden to take every instance of the ASCII character 'F' found in the input matrix and replace it
 * with the string '+F[F]'. the output string is lengthened as a result, so that the first four generations of an L-system with an axiom
 * of 'F' (as a starting point) would look like this:
 *
 * F
 * +F[F]
 * ++F[F][+F[F]]
 * +++F[F][+F[F]][++F[F][+F[F]]]
 *
 * more complex models can be created by adding multiple productions, or by introducing context matching into the L-system grammar.
 * for example, the message:
 *
 * 'production G F * +F'
 *
 * instructs jit.linden to replace values of 'F' with '+F' only if the character preceding the 'F' is a 'G'. otherwise, the 'F' is
 * simply echoed to the output string unchanged. if you were to give jit.linden the message:
 *
 * 'production G F * +F * G * G-'
 *
 * multiple productions are defined. 'F' is replaced as in the example above, but the character 'G' is now replaced by 'G-' as well.
 * up to 50 productions in the format <left_context> <strict_predecessor> <right_context> <successor_string> can be sent to jit.linden.
 * the predecessor symbol and the left and right contexts must be single characters in jit.linden (though this is not necessarily true
 * for all L-system interpreters). the successor (or replacement string) can be as long as you like.
 *
 * typically, the output of jit.linden will then be interpreted by an object which scans the string and interprets the symbols as commands.
 * the jit.turtle object interprets L-systems as turtle graphics, so that characters such as 'F', '+', and '-' acquire special meaning.
 * you could easily use jit.iter (or a holding jit.matrix object polled by 'getcell' messages) to access the Lindenmayer string in Max.
 * you can then 'select' against certain ASCII values and use those values to generate any kind of graphical or sonic data you like. the
 * example patches in the jit distribution called 'Lindenmayer Examples' give examples of generating 2-D graphics with L-systems within
 * Max/jit.
 */

#include "jit.common.h"

typedef struct _jit_linden
{
    t_object                ob;
    t_symbol                *ignore[128], *prodsym[200];
    char                    wildcard, leftbranch, rightbranch;
    long                    ignorecount, prodcount, boundmode;
} t_jit_linden;

void *_jit_linden_class;

t_jit_linden *_jit_linden_new(void);
void jit_linden_free(t_jit_linden *x);
t_jit_err jit_linden_matrix_calc(t_jit_linden *x, void *inputs, void *outputs);
void jit_linden_calculate_ndim(t_jit_linden *x, long dimcount, long *dim, long planecount,
    t_jit_matrix_info *in_mininfo, char *bip, t_jit_matrix_info *out_mininfo, char *bop);

```



```

t_jit_err jit_linden_init(void);

t_jit_err jit_linden_init(void)
{
    long attrflags=0;
    t_jit_object *attr, *mop;

    _jit_linden_class = jit_class_new("jit_linden", (method)jit_linden_new, (method)jit_linden_free,
        sizeof(t_jit_linden), A_CANT, 0L); //A_CANT = untyped

    //add mop
    mop = jit_object_new(_jit_sym_jit_mop, 1, 1); // #inputs, #outputs
    jit_mop_single_type(mop, _jit_sym_char);
    jit_mop_single_planecount(mop, 1);
    jit_class_addadornment(_jit_linden_class, mop);

    //add methods
    jit_class_addmethod(_jit_linden_class, (method)jit_linden_matrix_calc, "matrix_calc", A_CANT, 0L);

    //add attributes
    attrflags = JIT_ATTR_GET_DEFER_LOW | JIT_ATTR_SET_USURP_LOW;

    // ignore -- sets symbols to ignore while context matching
    attr = jit_object_new(_jit_sym_jit_attr_offset_array, "ignore", _jit_sym_symbol, 128,
        attrflags, (method)0L, (method)0L, calcoffset(t_jit_linden, ignorecount),
        calcoffset(t_jit_linden, ignoresym));
    jit_class_addattr(_jit_linden_class, attr);

    // production -- defines production rules for the L-system
    attr = jit_object_new(_jit_sym_jit_attr_offset_array, "production", _jit_sym_symbol, 200,
        attrflags, (method)0L, (method)0L, calcoffset(t_jit_linden, prodcnt),
        calcoffset(t_jit_linden, prodsym));
    jit_class_addattr(_jit_linden_class, attr);

    // wildcard -- sets wildcard character
    attr = jit_object_new(_jit_sym_jit_attr_offset, "wildcard", _jit_sym_char, attrflags,
        (method)0L, (method)0L, calcoffset(t_jit_linden, wildcard));
    jit_class_addattr(_jit_linden_class, attr);

    // leftbranch -- sets left branch character
    attr = jit_object_new(_jit_sym_jit_attr_offset, "leftbranch", _jit_sym_char, attrflags,
        (method)0L, (method)0L, calcoffset(t_jit_linden, leftbranch));
    jit_class_addattr(_jit_linden_class, attr);

    // rightbranch -- sets right branch character
    attr = jit_object_new(_jit_sym_jit_attr_offset, "rightbranch", _jit_sym_char, attrflags,
        (method)0L, (method)0L, calcoffset(t_jit_linden, rightbranch));
    jit_class_addattr(_jit_linden_class, attr);

    // boundmode -- sets wrapping flag
    attr = jit_object_new(_jit_sym_jit_attr_offset, "boundmode", _jit_sym_long, attrflags,
        (method)0L, (method)0L, calcoffset(t_jit_linden, boundmode));
    jit_class_addattr(_jit_linden_class, attr);

    jit_class_register(_jit_linden_class);

    return JIT_ERR_NONE;
}

t_jit_err jit_linden_matrix_calc(t_jit_linden *x, void *inputs, void *outputs)
{
    t_jit_err err=JIT_ERR_NONE;
    long in_savelock, out_savelock, dimmode;
    t_jit_matrix_info in_minfo, out_minfo;
    char *in_bp, *out_bp;
    long i, dimcount, planecount, dim[JIT_MATRIX_MAX_DIMCOUNT];
    void *in_matrix, *out_matrix;

    in_matrix = jit_object_method(inputs, _jit_sym_getindex, 0);
    out_matrix = jit_object_method(outputs, _jit_sym_getindex, 0);

    if (x&&in_matrix&&out_matrix) {

        in_savelock = (long) jit_object_method(in_matrix, _jit_sym_lock, 1);
        out_savelock = (long) jit_object_method(out_matrix, _jit_sym_lock, 1);

        jit_object_method(in_matrix, _jit_sym_getinfo, &in_minfo);
        jit_object_method(out_matrix, _jit_sym_getinfo, &out_minfo);
    }
}

```

```

jit_object_method(in_matrix,_jit_sym_getdata,&in_bp);
jit_object_method(out_matrix,_jit_sym_getdata,&out_bp);

if (!in_bp) { err=JIT_ERR_GENERIC; goto out;}
if (!out_bp) { err=JIT_ERR_GENERIC; goto out;}

//compatible types?
if ((in_minfo.type!=$_jit_sym_char)||(in_minfo.type!=out_minfo.type)) {
    err=JIT_ERR_MISMATCH_TYPE;
    goto out;
}

//compatible planes?
if ((in_minfo.planecount!=1)||(out_minfo.planecount!=1)) {
    err=JIT_ERR_MISMATCH_PLANE;
    goto out;
}

//compatible dimcounts?
if ((in_minfo.dimcount!=1)||(out_minfo.dimcount!=1)) {
    err=JIT_ERR_MISMATCH_DIM;
    post("needs to be 1 dim");
    goto out;
}

//get dimensions/planecount
dimcount = out_minfo.dimcount;
planecount = out_minfo.planecount;
for (i=0;i<dimcount;i++) {
    dim[i] = MIN(in_minfo.dim[i],out_minfo.dim[i]);
}

//calculate
jit_linden_calculate_ndim(x, dimcount, dim, planecount, &in_minfo, in_bp, &out_minfo, out_bp);
} else {
    return JIT_ERR_INVALID_PTR;
}

out:
jit_object_method(out_matrix,gensym("lock"),out_savelock);
jit_object_method(in_matrix,gensym("lock"),in_savelock);
return err;
}

//
//recursive functions to handle higher dimension matrices, by processing 2D sections at a time
//

// jit_linden_calculate_ndim() -- when x->dimmode==-1, sorts both dimensions together
void jit_linden_calculate_ndim(t_jit_linden *x, long dimcount, long *dim, long planecount,
    t_jit_matrix_info *in_minfo, char *bip, t_jit_matrix_info *out_minfo, char *bop)
{
    long i,j,k,l,p,width,height, index;
    long glow[3], outpix, lum, tol, bw, tmax, temp, mode;
    float indperc;
    uchar *ip,*op,*edge,*scanptr;
    t_symbol *tempsym;
    t_symbol *prodsym[200];
    long ignorecount=x->ignorecount;
    long prodcnt=x->prodcnt;
    long matchlen[200];
    long ismatch;
    char ignorebuf[256];
    char *tempchar;
    long cl_ok, cr_ok, level;
    long boundmode = CLAMP(x->boundmode, 0, 1);

    // reserved lindenmayer symbols as ASCII values
    char leftbranch=x->leftbranch; // '[' -- starts a branch in the L-system
    char rightbranch=x->rightbranch; // ']' -- ends a branch in the L-system
    char wildcard=x->wildcard; // '*' -- used as a wildcard character in context-matching

    // symbols to ignore
    for(i=0;i<256;i++) {
        ignorebuf[i] = 0;
    }
    for(i=0;i<ignorecount;i++) {

```

```

        tempsym=x->ignoresym[i];
        ignorebuf[*tempsym->s_name] = 1;
    }

// left context, strict predecessor, right context, successor, length of successor
for(i=0;i<prodcoun-3;i+=4) {
    prodsym[i]=x->prodsym[i];
    prodsym[i+1]=x->prodsym[i+1];
    prodsym[i+2]=x->prodsym[i+2];
    prodsym[i+3]=x->prodsym[i+3];
    matchlen[i] = strlen(prodsym[i+3]->s_name);
}

if (dimcount<1) return; //safety

    dim[1]=1;

    width = dim[0];
    height = dim[1];
    edge = bop+width-1; // pointer to end of row
    ip = bip;
    op = bop;
    // go through 1 dim only
    if(boundmode) *op++ = *ip++;
    // zero pad the start of the string so there's no boundary condition when context matching (faster)
    for (j=boundmode;j<width;j++) {
        ismatch=-1;
        cl_ok = 0;
        cr_ok = 0;
        // compare the current input char against the production predecessors.
        // if there's a match, set ismatch to which production it is (productions increment in sets of 4.
        for(k=0;k<prodcoun-3;k+=4) {
            if(*ip==*prodsym[k+1]->s_name) {
                ismatch=k;
                //post("match on %i production %i", j, k);

                // there's a match for this character. check its context to see if there's a true match

                // right context:
                if(*prodsym[ismatch+2]->s_name != wildcard) {
                    // need to check right context
                    //post("checking right context for %i", j);
                    scanptr=ip; // temporary pointer for checking context
                    cr_ok=0;
                    if(scanptr==edge) goto outcr; // end of line, there's no context match
                    scanptr++; // go right one cell and start checking context
                    while(1) {
                        if(ignorebuf[*scanptr]) { // cell is on the ignore list

                            if(scanptr==edge) goto outcr;
                            // end of line, there's no context match
                            scanptr++;
                        }
                        // keep going right until we're not in an ignored cell
                    }
                    else if(*scanptr==leftbranch) {
                        // we've hit a branch... need to find the closure

                        if(scanptr==edge) goto outcr;
                        // end of line, there's no context match
                        level = 1;
                        scanptr++;
                        while(level) {

                            if(*scanptr==leftbranch) {
                                // we've hit another branch that we need to get
                                through

                                    ++level;

                            }

                            if(*scanptr==rightbranch) {
                                // we've closed a branch, so go down a level

                                    --level;

                            }

                        }

                        if(scanptr==edge) goto outcr;
                        // end of line, there's no context match
                    }
                }
            }
        }
    }

```

```

        scanptr++;
    }
}
else if(*scanptr==*prodsym[ismatch+2]->s_name) {
    //post("right context is okay for %i", j);
    cr_ok=1; // right context okay for a match
    goto outcr; // get out of this loop
}
else {
    //post("right context is not okay for %i", j);
    cr_ok=0; // right context not okay
    goto outcr; // get out of this loop
}
}
}
else {
    // post("right context is okay");
    cr_ok=1; // right context is okay
}
outcr: // out of right context match
;

// left context:
if(*prodsym[ismatch]->s_name != wildcard) {
    // need to check left context
    //post("checking left context for %i", j);
    //post("needs to be %c", *prodsym[ismatch]->s_name);
    scanptr=ip; // temporary pointer for checking context
    //post("scanptr is %c", *scanptr);
    cl_ok=0;
    if((scanptr==bip)&&(*scanptr==*prodsym[ismatch]->s_name)) {
        // in boundmode 0, first cell is its own left context
        //post("match on first cell");
        cl_ok=1;
        goto outcl;
    }
    if(scanptr==bip) goto outcl;
    // beginning of line, there's no context match
    scanptr--; // go left one cell and start checking context
    while(1) {
        if(ignorebuf[*scanptr]) { // cell is on the ignore list

            if(scanptr==bip) goto outcl;
            // beginning of line, there's no context match
            scanptr--; // keep going left until we're not in an ignored cell
        }
        else if(*scanptr==rightbranch) {
            // we've hit a branch... need to find the closure
            if(scanptr==bip) goto outcl;
            // end of line, there's no context match
            level = 1;
            scanptr--;
            while(level) {

                if(*scanptr==rightbranch) {
                    // we've hit another branch that we need to get through

                    ++level;

                }

                if(*scanptr==leftbranch) {
                    // we've closed a branch, so go down a level

                    --level;

                }

                if(scanptr==bip) goto outcl;
                // end of line, there's no context match

                scanptr--;

            }
        }
    }
}

```

```

else if(*scanptr==*prodsym[ismatch]->s_name) {
    //post("left context is okay for %i", j);
    cl_ok=1; // left context okay for a match
    goto outcl; // get out of this loop
}
else {
    //post("left context is not okay for %i", j);
    cl_ok=0; // left context not okay
    goto outcl; // get out of this loop
}
}
}
else {
    // post("left context is okay");
    cl_ok=1; // left context is okay
}
outcl: // out of left context match
;

if(cr_ok&&cl_ok) goto substitute; // bail on the first true match, otherwise check next production
}
}

substitute:
;
//
// do the substitution (or not)
//

if(cr_ok&&cl_ok) {
    // contexts are okay... substitute the successor for the predecessor in the output matrix
    //post("match on %s on %i", prodsym[ismatch+1]->s_name, j);
    for(p=0;p<matchlen[ismatch];p++) {
        if(op<edge) { // avoid end of line
            *op++ = *(prodsym[ismatch+3]->s_name + p);
        }
        if(op==edge) { // last character
            *op = *(prodsym[ismatch+3]->s_name + p);
            goto endofline;
        }
    }
}
else {
    // contexts aren't okay... echo the input character to the output matrix
    if(op<edge) {
        *op++ = *ip;
    }
    if(op==edge) {
        *op=*ip;
        goto endofline;
    }
}

    *ip++;
endofline:
;
}

t_jit_linden *jit_linden_new(void)
{
    t_jit_linden *x;
    short i;

    if (x=(t_jit_linden *)jit_object_alloc(_jit_linden_class)) {
        x->ignorecount=0;
        x->prodcount=0;
        x->boundmode=1;
        // reserved lindenmayer symbols as ASCII values
        x->leftbranch=91; // '[' -- starts a branch in the L-system
        x->rightbranch=93; // ']' -- ends a branch in the L-system
        x->wildcard=42; // '*' -- used as a wildcard character in context-matching
    } else {
        x = NULL;
    }
    return x;
}

```

```
void jit_linden_free(t_jit_linden *x)
{
    //nada
}
```

max.jit.turtle.c

```

/*
    Copyright 2002 - Cycling '74
    R. Luke DuBois luke@cycling74.com
*/

/*
    jit.turtle emulates simple LOGO-style graphics.  it interprets single char values in its inlet as ASCII letters
    * that describe turtle graphics commands.  a valid ASCII input will generate appropriate drawing commands for use with
    * either the Max LCD object or the jit.lcd object (it potentially works with udraw and 242.qd as well).
    *
    * jit.turtle supports branching (via the '[' and ']') symbols up to a maximum stack size of 1024 nested branches.  while
    * this is kind of neat by itself, it is most useful when used with automatic string generators (either Max patches or objects
    * such as jit.linden).  ASCII codes which don't make any sense to jit.turtle are ignored.  symbols which the user wishes to
    * bypass can be filtered out by placing a 'select' or 'route' object with the appropriate ASCII numbers between the
    * source of the commands and the jit.turtle object.
    *
    * while jit.turtle doesn't use jit matrices to store and generate data, it uses the jit attribute system to set and poll
    * its internal state (origin, angle, scale, and clearmode are all jit attributes).  an ordinary 'turtle' Max object,
    * which doesn't require jit to run, could be ported from this code with a minimum of fuss.
    *
    * turtle graphics packages are fairly ubiquitous and some have conflicting syntaxes.  the turtle syntax used in this object
    * is optimized for the visualization of Lindenmayer Systems (or L-systems), which tend to use the same bunch of commands.
    * if the turtle you're used to has a different symbol table (e.g. penup/pendown independent of motion) or contains additional
    * commands, you should be able to add them under the max_jit_turtle_int() function easily.  things like additional algorithmic
    * support of variable angles and scaling (size of the steps in the 'forward' commands), polygon support (some turtle systems
    * have automatic polygon generation using 'G', 'g', '{', and '}'), and QuickDraw colors would be kind of fun, i think.
    *
*/

/*
    * if you were never subjected to turtle graphics in school (or elsewhere), the basic idea is this:
    *
    * you are a turtle.  as if that weren't bad enough, you are also inhibited by only being able to turn and go forward in
    * discrete amounts (you can also turn around).  however, to make up for this your creator has given you two very cool
    * genetic advantages over the average reptile:
    *
    * 1) you have a limitless supply of ink which secretes from your tail (which is usually referred to as the PEN).  by dragging
    * your tail along the floor as you move and turn you can therefore tell the entire world (or, at least, anyone who looks) all
    * about how much it sucks to be a turtle.
    *
    * 2) you can make quantum jumps back and forth to geographical positions and bearings where you've been before (though, to be
    * fair, you don't really know anything about those positions until you get there).  this is called BRANCHING, and it lets you,
    * a mere turtle, draw some incredibly complex images.
    *
    * as a turtle, your basic instruction set is the following (at least this is how i've implemented it):
    *
    * you can...
    *
    * - move forward a discrete amount with your tail down (drawing a line) -- the command for this is 'F'.
    * - move forward a discrete amount with your tail up (changing your position without drawing ('f')).
    * - turn right ('+'), left ('-'), or around ('!').
    * - press harder ('#') or lighter ('!') with your tail, creating a thicker or thinner line.
    * - start ('[') or end (']') a branch, which lets you remember a position and angle and then magically return to it later.
    *
    * your starting position and quantifiable attributes (such as how many steps you take when you move forward and how many
    * degrees you turn at a time) can be changed as you go by the Max patch in which you live.  good luck.
    *
*/

#include "jit.common.h"
#include "math.h" // necessary for the polar->cartesian stuff in the drawing routines

#define PI2 6.2831853

#define MAXSTACK 1024 // maximum number of branches... you can change this if you want to roll your own

typedef struct _max_jit_turtle
{
    t_object          ob;
    void              *obex; // we don't really need this, but whatever
    void              *turtleout;
    // turtle attributes -- feel free to add your own here.
    long              command;
    long              clearmode;
    long              origin[2];
    long              origincount;
    long              angle;
    long              scale;
}

```

```

double                thisangle[MAXSTACK];
long                 curstack;
long                 stacknew;
long                 pensize[MAXSTACK];
long                 stack_x[MAXSTACK], stack_y[MAXSTACK];

} t_max_jit_turtle;

void *max_jit_turtle_new(t_symbol *s, long argc, t_atom *argv);
void max_jit_turtle_free(t_max_jit_turtle *x);
void max_jit_turtle_assist(t_max_jit_turtle *x, void *b, long m, long a, char *s);
void max_jit_turtle_bang(t_max_jit_turtle *x); // does nothing
void max_jit_turtle_int(t_max_jit_turtle *x, long n); // this is where the QD stuff is interpreted
void max_jit_turtle_reset(t_max_jit_turtle *x); // resets the turtle's state

void *max_jit_turtle_class;

void main(void)
{
    long attrflags;
    void *p,*attr;

    setup(&max_jit_turtle_class, max_jit_turtle_new, (method)max_jit_turtle_free, (short)sizeof(t_max_jit_turtle),
        0L, A_GIMME, 0);

    p = max_jit_classex_setup(calcoffset(t_max_jit_turtle,obex));

    attrflags = JIT_ATTR_GET_DEFER_LOW | JIT_ATTR_SET_USURP_LOW ;

    // origin -- where to start drawing from (or reset to with a 'reset' message)
    attr = jit_object_new(_jit_sym_jit_attr_offset_array,"origin",_jit_sym_long,2,attrflags,
        (method)0,(method)0,calcoffset(t_max_jit_turtle,origincount),calcoffset(t_max_jit_turtle,origin));
    max_jit_classex_addattr(p,attr);

    // angle -- angle factor for turning the turtle
    attr = jit_object_new(_jit_sym_jit_attr_offset,"angle",_jit_sym_long,attrflags,
        (method)0,(method)0,calcoffset(t_max_jit_turtle,angle));
    max_jit_classex_addattr(p,attr);

    // scale -- stepsize for moving the turtle
    attr = jit_object_new(_jit_sym_jit_attr_offset,"scale",_jit_sym_long,attrflags,
        (method)0,(method)0,calcoffset(t_max_jit_turtle,scale));
    max_jit_classex_addattr(p,attr);

    // clearmode -- send a clear on reset or not
    attr = jit_object_new(_jit_sym_jit_attr_offset,"clearmode",_jit_sym_long,attrflags,
        (method)0,(method)0,calcoffset(t_max_jit_turtle,clearmode));
    max_jit_classex_addattr(p,attr);

    address((method)max_jit_turtle_reset,          "reset",          A_GIMME,0);
    max_jit_classex_standard_wrap(p,NULL,0);
    address((method)max_jit_turtle_assist,        "assist",          A_CANT,0);
    addbang((method)max_jit_turtle_bang);
    addint((method)max_jit_turtle_int);

    max_jit_class_addmethods(jit_class_findbyname(gensym("jit_turtle")));
}

void max_jit_turtle_bang(t_max_jit_turtle *x)
{
    // pontificate here...
    post("HEY, YOU! STOP BANGING THE TURTLE!!!");
}

void max_jit_turtle_int(t_max_jit_turtle *x, long n)
{
    t_atom a[16];
    double tempangle;
    double temp_x, temp_y;
    long dest_x, dest_y;
    long curstack = x->curstack;

    x->command = n; // why do we do this? i can't remember...

    // check to see if the integer received matches the ASCII code for one of these commands.
    // if so, compute the appropriate QuickDraw response and pass it out the outlet as a Max message.
    switch(n) {
        case (35): // '#' - increase pen size
            x->pensize[curstack] = x->pensize[curstack]+1;

```



```

        jit_atom_setlong(&a[0],x->pensize[curstack]);
        jit_atom_setlong(&a[1],x->pensize[curstack]);
        outlet_anything(x->turtleout, gensym("pensize"), 2, a);
        break;
    case (33): // '!' - decrease pen size
        if(x->pensize[curstack]>1) x->pensize[curstack] = x->pensize[curstack]-1;
        jit_atom_setlong(&a[0],x->pensize[curstack]);
        jit_atom_setlong(&a[1],x->pensize[curstack]);
        outlet_anything(x->turtleout, gensym("pensize"), 2, a);
        break;
    case (70): // 'F' - move forward and draw
        if(x->stacknew) {
            x->stack_x[curstack] = x->origin[0];
            x->stack_y[curstack] = x->origin[1];
            x->stacknew=0;
        }
        temp_x = (double)x->scale*jit_math_cos(x->thisangle[curstack]);
        temp_y = (double)x->scale*jit_math_sin(x->thisangle[curstack]);
        dest_x = x->stack_x[curstack]+temp_x+0.5;
        dest_y = x->stack_y[curstack]+temp_y+0.5;
        jit_atom_setlong(&a[0],x->stack_x[curstack]);
        jit_atom_setlong(&a[1],x->stack_y[curstack]);
        jit_atom_setlong(&a[2],dest_x);
        jit_atom_setlong(&a[3],dest_y);
        outlet_anything(x->turtleout, gensym("linesegment"), 4, a);
        x->stack_x[curstack] = dest_x;
        x->stack_y[curstack] = dest_y;
        break;
    case (102): // 'f' - move forward and don't draw
        if(x->stacknew) {
            x->stack_x[curstack] = x->origin[0];
            x->stack_y[curstack] = x->origin[1];
            x->stacknew=0;
        }
        temp_x = (double)x->scale*jit_math_cos(x->thisangle[curstack]);
        temp_y = (double)x->scale*jit_math_sin(x->thisangle[curstack]);
        dest_x = x->stack_x[curstack]+temp_x+0.5;
        dest_y = x->stack_y[curstack]+temp_y+0.5;
        x->stack_x[curstack] = dest_x;
        x->stack_y[curstack] = dest_y;
        break;
    case (91): // '[' - start a branch
        if(x->curstack>(MAXSTACK-2)) { // you can uncomment this post() if you prefer... it's kind of annoying, IMHO.
            // post("out of stack range -- not branching");
        }
        else {
            // copy current coords and angle to next branch and increment the stack
            x->stack_x[curstack+1] = x->stack_x[curstack];
            x->stack_y[curstack+1] = x->stack_y[curstack];
            x->thisangle[curstack+1] = x->thisangle[curstack];
            x->curstack++;
        }
        break;
    case (93): // ']' - end a branch and decrement the stack
        if(curstack>0) x->curstack--;
        break;
    case (43): // '+' - turn right
        x->thisangle[curstack]+=((x->angle/360.)*PI2);
        break;
    case (45): // '-' - turn left
        x->thisangle[curstack]-=((x->angle/360.)*PI2);
        break;
    case (124): // '|' - turn around
        x->thisangle[curstack]+=(0.5*PI2);
        break;
    default: // no match, don't do anything
        break;
}

}

void max_jit_turtle_reset(t_max_jit_turtle *x)
{
    t_atom a[16];
    short i;

    x->curstack = 0;
    x->stacknew = 1;

    for(i=0;i<MAXSTACK;i++) {
        x->thisangle[i] = -PI2/4.; // start facing north (upwards, towards your menubar).
    }
}

```

```

        x->stack_x[i] = x->origin[0];
        x->stack_y[i] = x->origin[1];
        x->pensize[i]=1;
    }

    jit_atom_setlong(&a[0],x->pensize[x->curstack]);
    jit_atom_setlong(&a[1],x->pensize[x->curstack]);
    outlet_anything(x->turtleout, gensym("pensize"), 2, a);

    // if the 'clearmode' attribute is set have jit.turtle tell the QuickDraw object downstream to clear itself.
    if(x->clearmode) outlet_anything(x->turtleout, gensym("clear"),0,0L);
}

void max_jit_turtle_assist(t_max_jit_turtle *x, void *b, long m, long a, char *s)
{
    //nada for now
}

void max_jit_turtle_free(t_max_jit_turtle *x)
{
    //only max object, no jit object
    max_jit_obex_free(x);
}

void *max_jit_turtle_new(t_symbol *s, long argc, t_atom *argv)
{
    t_max_jit_turtle *x;
    t_jit_matrix_info info;
    long attrstart,i,j;
    jit_matrix_info_default(&info);

    x = (t_max_jit_turtle *)max_jit_obex_new(max_jit_turtle_class,gensym("jit_turtle"));
    max_jit_obex_dumpout_set(x, outlet_new(x,0L)); //general purpose outlet(rightmost)
    x->turtleout = outlet_new(x,0L); // outlet for the LCD commands

    x->clearmode = 1;

    x->scale = 10;
    x->angle = 30;
    x->origin[0] = 80; // kind of arbitrary, but i think the default jit.lcd matrix size is 160x120.
    x->origin[1] = 120;
    x->origincount = 2;
    x->curstack=0;
    x->stacknew=1;
    for(i=0;i<MAXSTACK;i++) {
        x->thisangle[i] = -PI2/4.; // start facing north (upwards)
        x->stack_x[i] = x->origin[0];
        x->stack_y[i] = x->origin[1];
        x->pensize[i] = 1;
    }

    max_jit_attr_args(x,argc,argv); //handle attribute args

out:
    return (x);
}

```

jit.lindenpoly.c

```

/*
    Copyright 2003 - Cycling '74
    R. Luke DuBois luke@music.columbia.edu
*/

/*
 * jit.lindenpoly sorts out a 1-dim L-system into 2-dims according to branches.
 *
 */

#define MAXSTACK 1024

#include "jit.common.h"

typedef struct _jit_lindenpoly
{
    t_object                                ob;
    char                                    leftbranch, rightbranch;
    long                                    boundmode;
} t_jit_lindenpoly;

void *_jit_lindenpoly_class;

t_jit_lindenpoly *_jit_lindenpoly_new(void);
void jit_lindenpoly_free(t_jit_lindenpoly *x);
t_jit_err jit_lindenpoly_output_adapt(void *mop, void *mop_io, void *matrix); // need to add a second dim to the output matrix
t_jit_err jit_lindenpoly_matrix_calc(t_jit_lindenpoly *x, void *inputs, void *outputs);
void jit_lindenpoly_calculate_ndim(t_jit_lindenpoly *x, long dimcount, long *dim, long planeccount,
    t_jit_matrix_info *in_mininfo, char *bip, t_jit_matrix_info *out_mininfo, char *bop);
t_jit_err jit_lindenpoly_init(void);

t_jit_err jit_lindenpoly_init(void)
{
    long attrflags=0;
    t_jit_object *o, *attr, *mop;

    _jit_lindenpoly_class = jit_class_new("jit_lindenpoly", (method)jit_lindenpoly_new, (method)jit_lindenpoly_free,
        sizeof(t_jit_lindenpoly), A_CANT, 0L); // A_CANT = untyped

    //add mop
    mop = jit_object_new(_jit_sym_jit_mop, 1, 1); // #inputs, #outputs
    o = jit_object_method(mop, _jit_sym_getoutput, 1);
    jit_object_method(o, _jit_sym_ioproc, jit_lindenpoly_output_adapt);

    jit_class_addadornment(_jit_lindenpoly_class, mop);

    //add methods
    jit_class_addmethod(_jit_lindenpoly_class, (method)jit_lindenpoly_matrix_calc, "matrix_calc",
        A_CANT, 0L);

    //add attributes
    attrflags = JIT_ATTR_GET_DEFER_LOW | JIT_ATTR_SET_USURP_LOW;

    // leftbranch -- sets left branch character
    attr = jit_object_new(_jit_sym_jit_attr_offset, "leftbranch", _jit_sym_char, attrflags,
        (method)0L, (method)0L, calcoffset(t_jit_lindenpoly, leftbranch));
    jit_class_addattr(_jit_lindenpoly_class, attr);

    // rightbranch -- sets right branch character
    attr = jit_object_new(_jit_sym_jit_attr_offset, "rightbranch", _jit_sym_char, attrflags,
        (method)0L, (method)0L, calcoffset(t_jit_lindenpoly, rightbranch));
    jit_class_addattr(_jit_lindenpoly_class, attr);

    // boundmode -- sets wrapping flag
    attr = jit_object_new(_jit_sym_jit_attr_offset, "boundmode", _jit_sym_long, attrflags,
        (method)0L, (method)0L, calcoffset(t_jit_lindenpoly, boundmode));
    jit_class_addattr(_jit_lindenpoly_class, attr);

    jit_class_register(_jit_lindenpoly_class);

    return JIT_ERR_NONE;
}

```

```

t_jit_err jit_lindenpoly_output_adapt(void *mop, void *mop_io, void *matrix)
{
    void *m;
    t_jit_matrix_info info;
    long tmp;

    if (matrix&&(m=jit_object_method(mop_io,_jit_sym_getmatrix))) {

        if(jit_attr_getlong(mop,_jit_sym_adapt)
           {
                jit_object_method(matrix,_jit_sym_getinfo,&info);
                info.dim[1] = MAXSTACK;
                info.dimcount = 2;
                jit_object_method(m,_jit_sym_setinfo,&info);
                jit_object_method(m,_jit_sym_clear);
            }
           else {
                jit_object_method(m,_jit_sym_clear);
            }
        }

        return JIT_ERR_NONE;
    }
}

t_jit_err jit_lindenpoly_matrix_calc(t_jit_lindenpoly *x, void *inputs, void *outputs)
{
    t_jit_err err=JIT_ERR_NONE;
    long in_savelock,out_savelock, dimmode;
    t_jit_matrix_info in_minfo,out_minfo;
    char *in_bp,*out_bp;
    long i,dimcount,planecount,dim[JIT_MATRIX_MAX_DIMCOUNT];
    void *in_matrix, *out_matrix;

    in_matrix = jit_object_method(inputs, _jit_sym_getindex, 0);
    out_matrix = jit_object_method(outputs, _jit_sym_getindex, 0);

    if (x&&in_matrix&&out_matrix) {

        in_savelock = (long) jit_object_method(in_matrix,_jit_sym_lock,1);
        out_savelock = (long) jit_object_method(out_matrix,_jit_sym_lock,1);

        jit_object_method(in_matrix,_jit_sym_getinfo,&in_minfo);
        jit_object_method(out_matrix,_jit_sym_getinfo,&out_minfo);

        jit_object_method(in_matrix,_jit_sym_getdata,&in_bp);
        jit_object_method(out_matrix,_jit_sym_getdata,&out_bp);

        if (!in_bp) { err=JIT_ERR_GENERIC; goto out;}
        if (!out_bp) { err=JIT_ERR_GENERIC; goto out;}

        //compatible types?
        if ((in_minfo.type!=_jit_sym_char)||(in_minfo.type!=out_minfo.type)) {
            err=JIT_ERR_MISMATCH_TYPE;
            goto out;
        }

        //compatible planes?
        if ((in_minfo.planecount!=1)||(out_minfo.planecount!=1)) {
            err=JIT_ERR_MISMATCH_PLANE;
            goto out;
        }

        //compatible dimcounts?
        if ((in_minfo.dimcount!=1)||(out_minfo.dimcount!=2)) {
            err=JIT_ERR_MISMATCH_DIM;
            post("needs to be 1 dim");
            goto out;
        }

        //get dimensions/planecount
        dimcount = out_minfo.dimcount;
        planecount = out_minfo.planecount;
        dim[0] = MIN(in_minfo.dim[0],out_minfo.dim[0]);
        dim[1] = out_minfo.dim[1];

        //calculate
        jit_lindenpoly_calculate_ndim(x, dimcount, dim, planecount, &in_minfo, in_bp, &out_minfo, out_bp);
    }
}

```

```

    } else {
        return JIT_ERR_INVALID_PTR;
    }

out:
    jit_object_method(out_matrix, gensym("lock"), out_savelock);
    jit_object_method(in_matrix, gensym("lock"), in_savelock);
    return err;
}

//
// recursive functions to handle higher dimension matrices, by processing 2D sections at a time
//

// jit_lindenpoly_calculate_ndim() -- when x->dimmode==1, sorts both dimensions together
void jit_lindenpoly_calculate_ndim(t_jit_lindenpoly *x, long dimcount, long *dim, long planeount,
    t_jit_matrix_info *in_minfo, char *bip, t_jit_matrix_info *out_minfo, char *bop)
{
    long i, j, k, l, p, width, height, index;
    float indperc;
    uchar *ip, *op, *edge, *scanptr;

    long stride;
    long stackpoint[MAXSTACK];
    long stackhistory[MAXSTACK];
    long currentvoice = 0;
    long curstack = 0;
    long tempstack;
    int newvoiceflag = 0;

    char *tempchar;
    long cl_ok, cr_ok, level;
    long boundmode = CLAMP(x->boundmode, 0, 1);

    // reserved lindenpolymayer symbols as ASCII values
    char leftbranch=x->leftbranch; // '[' -- starts a branch in the L-system
    char rightbranch=x->rightbranch; // ']' -- ends a branch in the L-system

    for(i=0; i<dim[1]; i++) // initialize stackpoint... used to find a free line
    {
        stackpoint[i] = -1;
        stackhistory[i] = 0;
    }
    stackpoint[0] = 0;

    if (dimcount < 1) return; // safety

    width = dim[0];
    height = dim[1];
    stride = out_minfo->dimstride[1];
    edge = bop+width-1; // pointer to end of row
    ip = bip;
    op = bop;

    // go through 1 dim only
    for (j=0; j<width; j++) {
        if((j>0)&&(*ip == 0)) goto endoffline; // EOF... we're outta here
        if (*ip == leftbranch) { // increment the stack to the next free line
            // find next free line
            tempstack = curstack;
            for(i=curstack; i<height; i++) {
                tempstack++;
                if(stackpoint[tempstack] < stackpoint[curstack])
                    stackpoint[tempstack] =
                        stackpoint[curstack]; // start new voice on top of current voice

                newvoiceflag = 1;
                curstack=tempstack;
                currentvoice++;
                stackhistory[currentvoice] =
                    curstack;

                goto foundfreeline;
            }
            post("out of free branches");
            foundfreeline:
            ;
            //post("left branch at %i", j);
        }
    }
}

```

```

else if (*ip == rightbranch) { // decrement the stack to the previous line
    if(--currentvoice<0) currentvoice = 0;
    curstack = stackhistory[currentvoice]; // move down to last

    //post("right branch at %i!", j);

}
else {
    if(newvoiceflag) {
        stackpoint[curstack]--; // KLUDGE!!!
        newvoiceflag=0;
    }
    op = bop + stackpoint[curstack] + curstack*stride;
    stackpoint[curstack]++;
    //post("stackpoint %i at stack %i!", stackpoint[curstack],

*op++ = *ip;
}
*ip++;
}
}
endifline:
;
}

t_jit_lindenpoly *_jit_lindenpoly_new(void)
{
    t_jit_lindenpoly *x;
    short i;

    if (x=(t_jit_lindenpoly *)jit_object_alloc(_jit_lindenpoly_class)) {
        x->boundmode=1;
        // reserved lindenmayer symbols as ASCII values
        x->leftbranch=91; // '[' -- starts a branch in the L-system
        x->rightbranch=93; // ']' -- ends a branch in the L-system
    } else {
        x = NULL;
    }
    return x;
}

void jit_lindenpoly_free(t_jit_lindenpoly *x)
{
    //nada
}

```

Bibliography

Books / Articles

L-systems, Grammar Models, and Computational Modeling

Bossomaier, Terry, and **Green**, David. Patterns in the Sand - Computers, Complexity and Life. Reading, MA: Perseus Books, 1998.

Chomsky, Noam. "Three Models for the Description of Language." IRE Transactions on Information Theory 2. Washington, DC: IEEE (formerly IRE), 1956.

Lindenmayer, Aristid. "Mathematical models for cellular interactions in development" (Two Parts). Journal of Theoretical Biology 18. New York: Elsevier, 1968.

Mandelbrot, B.B. The fractal geometry of nature. San Francisco: W.H. Freeman, 1982.

Minsky, Marvin, and **Papert**, Seymour. Perceptrons: an introduction to computational geometry. Cambridge, MA: MIT Press, 1969.

Prusinkiewicz, Przemyslaw. "Virtual plants: new perspectives for computer graphics." The ScienceTerrapin 4:8. London, Ontario: UWO Press, 1979.

Prusinkiewicz, Przemyslaw. "Score Generation with L-systems." Proceedings of the International Computer Music Conference, 1986. Den Haag: ICMA, 1986.

Prusinkiewicz, Przemyslaw, and **Lindenmayer**, Aristid. The Algorithmic Beauty of Plants. New York: Springer-Verlag, 1990.

Prusinkiewicz, Przemyslaw, **Hammel**, Mark, **Hanan**, Jim, and **Mech**, Radomír. "L-systems: From The Theory To Visual Models Of Plants." Proceedings of the Second CSIRO Symposium on Computational Challenges in Life Sciences. Brisbane, CSIRO Publishing, 1996.

Sierpinski, Waclaw. "The Sierpinski Triangle" (1916). Excerpted in Devaney, Keen, eds. Chaos and fractals : the mathematics behind the computer graphics. Providence, RI: American Mathematical Society, 1989.

Szilard, A. L., and **Quinton**, R. E. "An interpretation for DOL systems by computer graphics." The ScienceTerrapin 4:8. London, Ontario: UWO Press, 1979.

Taylor, C. E. "Fleshing Out." In Langton, Taylor, et al, eds. Artificial Life II. Redwood City, CA: Addison-Wesley, 1992.

Turing, Alan. "On computable numbers, with an application to the *Entscheidungsproblem*" (1936). Reprinted in M. Davis, Ed. The Undecidable: Basic Papers on Undecidable Propositions, Unsolvability Problems and Computable Functions. New York: Raven Press, 1965.

Wolfram, Stephen. A New Kind of Science. Champaign, IL: Wolfram Media, 2002.

Music Theory, Music Cognition, Acoustics, and Psychoacoustics

Bregman, Albert S. Auditory Scene Analysis. Cambridge, MA: MIT Press, 1990.

Foxley, Eric. "The Harmonisation of Melodies by Computer." Proceedings of the Second Symposium International Informatique et Musicologie. Paris: SIIM, 1982.

Lerdahl, Fred, and **Jackendoff**, Ray. A Generative Theory of Tonal Music. Cambridge, MA: MIT Press, 1983.

Lerdahl, Fred. Tonal Pitch Space. New York: Oxford University Press, 2001.

Spector, Lee, and **Alpern**, Adam. "Induction and Recapitulation of Deep Musical Structure." Proceedings of the IJCAI-95 Workshop On Artificial Intelligence and Music. Montréal: International Joint Conference on Artificial Intelligence, 1995.

Temperley, David. The Cognition of Basic Musical Structures. Cambridge, MA: MIT Press, 2001.

Computer Music and Interactivity

Chadabe, Joel. Electric Sound: The Past and Promise of Electronic Music. Upper Saddle River, NJ: Prentice Hall, 1996.

Garnett, Guy E. "The Aesthetics of Interactive Computer Music." Computer Music Journal 25:1. Cambridge, MA: MIT Press, 2001.

Johnson, Steven. Interface Culture: How New Technology Transforms the Way We Create and Communicate. New York: HarperCollins, 1997.

Maurer, John. "A Brief History of Algorithmic Composition." Unpublished. Stanford University. ccrma-www.stanford.edu/~blackrse/algorithm.html

Paradiso, Joseph A. "American Innovations in Electronic Musical Instruments." New Music Box 6. <http://www.newmusicbox.org/third-person/oct99/>

Rowe, Robert. Interactive Music Systems. Cambridge, MA: MIT Press, 1993.

Rowe, Robert. Machine Musicianship. Cambridge, MA: MIT Press, 2001.

Théberge, Paul. Any Sound You Can Imagine: Making Music / Consuming Technology. Hanover, NH: Wesleyan University Press, 1997.

Supper, Martin. "A Few Remarks on Algorithmic Composition." Computer Music Journal 25:1. Cambridge, MA: MIT Press, 2001.

Winkler, Todd. Composing Interactive Music: Techniques and Ideas Using Max. Cambridge, MA: MIT Press, 1998.

Software

Clayton, Joshua, **Jones**, Randall, **Bernstein**, Jeremy, **DuBois**, R. Luke, and **Grosse**, Darwin. Jitter. <http://www.cycling74.com/products/jitter.html>

Freed, Adrian, **Lee**, Michael, **Ellison**, Steve, and **Zicarelli**, David. The Max lcd object. www.cnmat.berkeley.edu/MAX

Gogins, Michael. Silence. www.pipeline.com/~gogins/

Goodall, Leigh, and **Watson**, Matthew. lsys2midi. <http://ironbark.bendigo.latrobe.edu.au/~soddell/lsys/mapping.htm>

Hiller, Lejaran, and **Baker**, Robert. MUSICOMP. Described at <http://arts.ucsc.edu/faculty/cope/history.html>

Olafsson, Kjartan. CALMUS. <http://www.listir.is/calmus/>

Oppenheim, Daniel. "DMIX: An Environment for Composition." Proceedings of the International Computer Music Conference, 1989. Columbus, OH: ICMA, 1989.

Puckette, Miller. "EXPLODE: A User Interface for Sequencing and Score Following." Proceedings of the International Computer Music Conference, 1990. Glasgow: ICMA, 1990.

Sharp, David. LMUSE. <http://www.geocities.com/Athens/Academy/8764/lmuse/lmuse.html>

Zicarelli, David et al. Max/MSP. <http://www.cycling74.com/products/maxmsp.html>

Music Recordings

Dodge, Charles. "Profile." Electro-acoustic music I. Acton, MA: Neuma Records, 1990.

Dodge, Charles. "Viola Elegy." Viola elegy [for] viola and tape. Lebanon, NH: Frog Peak Music, 1987.

Dodge, Charles. "Earth's Magnetic Field" (1987). Columbia-Princeton Electronic Music Center: 1961-1973. New York: New World Records, 1998.

Nelson, Gary Lee. "Summer Song" (1991) / "Goss" (1993). Available online at timara.con.oberlin.edu/~gnelson

Pope, Stephen Travis. "Day, an Improvisation" (1987). Available online at <http://www.create.ucsb.edu/Siren/Pieces/>

Roads, Curtis. "Sequence Symbols." New computer music. Mainz: Wergo, 1987.

Scores

Growing Pains (for mandolin and electronics)

Repeat After Me (for flute and electronics)

Biology I/II/III (for violin)

Video files and mp3 recordings of the pieces are available at:

<http://music.columbia.edu/~luke/dissertation>

All music copyright © 2003 R. Luke DuBois / The Freight Elevator Quartet, Inc. (ASCAP). All rights reserved.

R. Luke DuBois

Growing Pains

for mandolin and electronics

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Musical Arts
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY
2003

© 2003
Roger Luke DuBois
All Rights Reserved

Introduction

Growing Pains is a piece for mandolin accompanied by live processing provided by a computer, as well as live video generated by the performer. It was written in the autumn of 2002 for Terry Pender. It is submitted in partial fulfillment of the requirements for the degree of Doctor of Musical Arts in the Graduate School of Arts and Sciences, Columbia University, 2003.

The duration of the piece should be ~18 minutes.

Performance Notes

Equipment requirements:

- Macintosh PowerPC computer with stereo audio input and output (G3/500 or higher recommended).
- Pickup or microphone for the mandolin, mixer, and PA with 2 high-quality loudspeakers.
- High-quality SVGA video projection system.
- Hardware pitch-to-MIDI converter (e.g. Roland GI-10, IVL PitchRider) and appropriate MIDI interface hardware are optional, but may improve responsiveness on slower computers.

All software (Max patch and video files) is available from the composer upon request.

The mandolin should be amplified with a pickup (ideally) or a very close directional microphone. The mandolin and the computer output should be balanced in the PA so that the echoes generated by the computer don't overpower the dry mandolin signal. The piece is meant to have an 'electric' feel; the amplified sounds can be louder than the acoustic sound of the mandolin.

The piece relies to some degree on the ability of the computer to detect the pitch played by the mandolin performer. If a microphone is used, it is strongly recommended that the piece be tested in the performance space first, so that the ideal microphone placement for pitch-tracking can be achieved. If necessary, two microphones can be used: one for the computer input, and another for sound reinforcement.

Notation

All notation symbols in the score are standard. Accidentals carry through the measure. A second staff indicating cues in the signal processing is provided so that the performer (or a computer operator) can make sure that the computer is correctly following the performer.

The performer should play as 'clean' as possible without excessive staccato, i.e. notes on adjacent strings should not ring over one another.

Growing Pains

R. Luke DuBois

$\text{♩} = 90$ **A** with a steady rhythm throughout

Mandolin *mf*

Echo Cues

3

6

9

12

15

18



21



24



27



30



33



36

39

42

45

48

51

54 B

cue 3

57

60

63

66

69



90

cuc 4

93

96

99

102

105

108 C

cue 5

111

114

117

120

123 D

cue 6



147



150



153



156



159



162

E



165



Musical staff for measures 165-167. The staff is in treble clef with a key signature of two flats (B-flat and E-flat). It contains three measures of music, each featuring a complex rhythmic pattern of eighth and sixteenth notes with various rests and accents.

168



Musical staff for measures 168-170. The staff is in treble clef with a key signature of two flats. It contains three measures of music, continuing the rhythmic complexity from the previous staff.

171



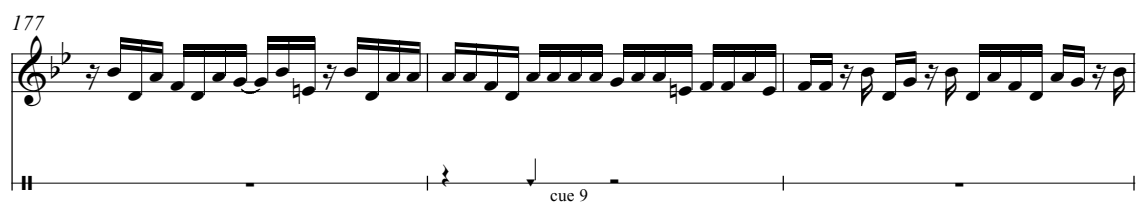
Musical staff for measures 171-173. The staff is in treble clef with a key signature of two flats. It contains three measures of music.

174



Musical staff for measures 174-176. The staff is in treble clef with a key signature of two flats. It contains three measures of music.

177



Musical staff for measures 177-179. The staff is in treble clef with a key signature of two flats. It contains three measures of music. Below the staff, there is a double bar line followed by a rest and the text "cue 9" centered under the rest.

180



Musical staff for measures 180-182. The staff is in treble clef with a key signature of two flats. It contains three measures of music.



204



207



210

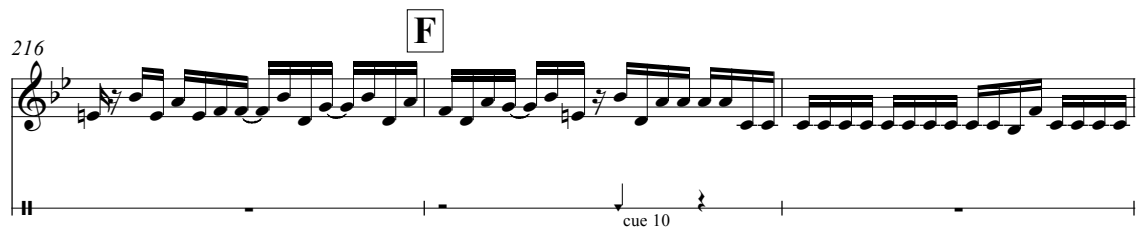


213



216

F



219






243



Musical notation for measures 243-245. The key signature has two flats (B-flat and E-flat). The melody consists of eighth and sixteenth notes with frequent rests.

246



Musical notation for measures 246-248. Measure 248 contains a double bar line and the text "cue 11" below it.

249



Musical notation for measures 249-251. The melody continues with eighth and sixteenth notes.

252



Musical notation for measures 252-254. The melody continues with eighth and sixteenth notes.

255



Musical notation for measures 255-257. The melody continues with eighth and sixteenth notes.

258



Musical notation for measures 258-260. The melody continues with eighth and sixteenth notes.

261

264

267

270

273 **G**

276



298



301



304



307



310



313



316



319

322

325

H

cue 13

328

331

334





379 I

cue 14

382

385

388

391

394



418

Musical staff 418: Treble clef, key signature of two flats, 4/4 time signature. The staff contains a sequence of eighth and sixteenth notes with various rests and ties.

421

Musical staff 421: Treble clef, key signature of two flats, 4/4 time signature. The staff contains a sequence of eighth and sixteenth notes. Below the staff is a cue line with a vertical line and the text "cue 15".

424

Musical staff 424: Treble clef, key signature of two flats, 4/4 time signature. The staff contains a sequence of eighth and sixteenth notes with various rests and ties.

427

Musical staff 427: Treble clef, key signature of two flats, 4/4 time signature. The staff contains a sequence of eighth and sixteenth notes with various rests and ties.

430

Musical staff 430: Treble clef, key signature of two flats, 4/4 time signature. The staff contains a sequence of eighth and sixteenth notes with various rests and ties.

433

Musical staff 433: Treble clef, key signature of two flats, 4/4 time signature. The staff contains a sequence of eighth and sixteenth notes with various rests and ties, ending with a double bar line.

R. Luke DuBois

Repeat After Me

for flute and electronics

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Musical Arts
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY
2003

© 2003
Roger Luke DuBois
All Rights Reserved

Introduction

Repeat After Me is a piece for flute accompanied by live processing provided by a computer, as well as live video generated by the performer. It was written in the winter of 2003 for Natacha Diels. It is submitted in partial fulfillment of the requirements for the degree of Doctor of Musical Arts in the Graduate School of Arts and Sciences, Columbia University, 2003.

The duration of the piece should be ~9 minutes.

Performance Notes

Equipment requirements:

- Macintosh PowerPC computer with stereo audio input and output (G4/667 or higher recommended).
- Microphone for the flute, mixer, and PA with 2 high-quality loudspeakers.
- High-quality SVGA video projection system.
- Hardware pitch-to-MIDI converter (e.g. Roland GI-10, IVL PitchRider) and appropriate MIDI interface hardware are optional, but may improve responsiveness on slower computers.

All software (Max patch, samples and video files) is available from the composer upon request. The piece involves heavy use of pre-recorded flute samples. Instructions are available from the composer on how to replace them with those of a different performer.

The flute should be amplified with a very close directional microphone. A wearable microphone (e.g. a reasonably good-quality Lavalier) should work well. The flute and the computer output in the PA speakers should be balanced so that the processing and samples generated by the computer don't overpower the live flute sound. The piece relies on the sonic integration of the live flute and accompaniment lines generated by the computer; in order to achieve a good balance, it may be necessary for the flute to be reinforced through the PA as well.

The piece relies to some degree on the ability of the computer to detect the pitch played by the flute performer. It is strongly recommended that the piece be tested in the performance space first, so that the ideal microphone placement for pitch-tracking can be achieved. If necessary, two microphones can be used: one for the computer input, and another for sound reinforcement.

Notation

All notation symbols in the score are standard. Accidentals carry through the measure. Two extra staves indicating cues in the accompaniment (as well as the outer voices of the sample clusters) are provided so that the performer (or a computer operator) can make sure that the computer is correctly following the performer.

Repeat After Me

R. Luke DuBois

♩ = 115 **A** lyrical, with an easy pacing

Flute *mf*

Cues ambient sound cue 1 begins

Samples (dynamics tied to flute)

8

13

p *mf*

17

21

mp

flute echoes fade in (ppp) (mf)

26

mf

30

mf

34

mf

granular processing cue 1

39

3

3

3

p

echoes: (mf) (pp)

43

3

3

3

3

granular processing and echoes off

47

3

3

3

3

mp

b

51

mp

3

3

b

56

Musical notation for measures 56-59. The top staff is a treble clef with a key signature of one flat. It contains a melodic line with triplets and sextuplets. The bottom staff is a grand staff with a bass clef, which is mostly empty.

60

Musical notation for measures 60-63. The top staff is a treble clef with a key signature of one flat. It contains a melodic line with triplets and sextuplets. The bottom staff is a grand staff with a bass clef, which is mostly empty.

64

B eerie and detached

Musical notation for measures 64-67. The top staff is a treble clef with a key signature of one flat. It contains a melodic line with a triplet and a long note. The bottom staff is a grand staff with a bass clef. There are annotations for "ambient sound cue 1 fades", "granular processing cue 2", and "ambient sound cue 2 begins". Dynamics "pp" and "mp" are also indicated.

71

Musical notation for measures 71-74. The top staff is a treble clef with a key signature of one flat. It contains a melodic line with various intervals and accidentals. The bottom staff is a grand staff with a bass clef, showing chordal accompaniment.

74

Musical score for measures 74-76. The top staff is a treble clef with a complex melodic line featuring many accidentals and slurs. The middle staff is a double bar line. The bottom staff is a bass clef with a simple harmonic accompaniment.

77

Musical score for measures 77-78. The top staff continues the melodic line with slurs and accents. The middle staff is a double bar line. The bottom staff continues the harmonic accompaniment.

79

Musical score for measures 79-80. The top staff features a melodic line with a slur and an accent. The middle staff is a double bar line. The bottom staff continues the harmonic accompaniment.

81

Musical score for measures 81-82. The top staff continues the melodic line. The middle staff has a double bar line and the text "flute echoes (mp)" below it. The bottom staff continues the harmonic accompaniment.

83

Musical score for measures 83-84. The top staff is a treble clef with a key signature of one flat (B-flat). The melody consists of eighth and sixteenth notes with various accidentals. The bottom staff is a bass clef with a key signature of one flat, containing a few notes and rests.

85

Musical score for measures 85-87. The top staff is a treble clef with a key signature of one flat. It features a melodic line with a *mf* dynamic marking and a slur over the final two measures. The bottom staff is a bass clef with a key signature of one flat, showing a few notes and rests.

88

Musical score for measures 88-89. The top staff is a treble clef with a key signature of one flat. The melody starts with a *mp* dynamic marking. The bottom staff is a bass clef with a key signature of one flat, containing a few notes and rests.

90

Musical score for measures 90-91. The top staff is a treble clef with a key signature of one flat. The melody includes a *granular processing off* instruction. The bottom staff is a bass clef with a key signature of one flat, containing a few notes and rests.

92

Musical score for measures 92-93. The top staff is a treble clef with a key signature of one flat (B-flat). It contains a complex melodic line with many accidentals and slurs. The bottom staff is a bass clef with a key signature of one flat, containing a simple accompaniment line.

94

Musical score for measures 94-95. The top staff is a treble clef with a key signature of one flat. It features a melodic line with a dynamic marking *p* and a slur. The bottom staff is a bass clef with a key signature of one flat. Below the bottom staff, the text "echoes off" and "granular processing cue 3" is written.

98

Musical score for measures 98-99. The top staff is a treble clef with a key signature of one flat. It contains a melodic line with various accidentals and slurs. The bottom staff is a bass clef with a key signature of one flat. Below the bottom staff, the text "granular processing cue 4" is written.

101

Musical score for measures 101-102. The top staff is a treble clef with a key signature of one flat. It features a melodic line with slurs and accidentals. The bottom staff is a bass clef with a key signature of one flat, containing a simple accompaniment line.

104

Musical score for measures 104-106. The top staff is a treble clef with a complex melodic line featuring many accidentals. The middle staff is a grand staff with a whole rest. The bottom staff is a bass clef with a simple accompaniment line.

107

Musical score for measures 107-109. The top staff continues the complex melodic line. The middle staff has a whole rest. The bottom staff has a simple accompaniment line with some accidentals.

110

Musical score for measures 110-112. The top staff continues the melodic line. The middle staff has a whole rest and a "flute echoes (mf)" annotation. The bottom staff has a simple accompaniment line.

113

Musical score for measures 113-115. The top staff continues the melodic line. The middle staff has a whole rest and a "granular processing off" annotation. The bottom staff has a simple accompaniment line.

118

mf

122

126

p

ambient sound cue 2 fades

133 **C** expressive and lyrical

p

flute echoes off

138

ambient sound cue 3 begins

second sampler begins (mp)

142

second sampler (mp)

147

mf

(mf)

151

156

Musical score for measures 156-160. The top staff is a treble clef with a melody starting on G4, moving to A4, B4, C5, and ending with a trill on B4. The bottom staff is a bass clef with a simple accompaniment of chords: G2-B2, A2-C3, B2-D3, C3-E3, and B2-D3.

161

Musical score for measures 161-165. The top staff is a treble clef with a melody starting on G4, featuring a triplet of eighth notes (A4, B4, C5), followed by a descending line to E4 and ending on G4. The bottom staff is a bass clef with chords: G2-B2, A2-C3, B2-D3, C3-E3, and B2-D3.

165

Musical score for measures 165-170. The top staff is a treble clef with a melody starting on G4, moving to A4, B4, C5, and ending with a trill on B4. The bottom staff is a bass clef with chords: G2-B2, A2-C3, B2-D3, C3-E3, and B2-D3.

170

Musical score for measures 170-175. The top staff is a treble clef with a melody starting on G4, featuring a triplet of eighth notes (A4, B4, C5), followed by a descending line to E4 and ending on G4. The bottom staff is a bass clef with chords: G2-B2, A2-C3, B2-D3, C3-E3, and B2-D3. A dynamic marking *mp* is present in the bottom staff.

174

mf

178

mp

182

mp

186

mp pp mf

flute echoes (fade to mf)

190

Musical score for measures 190-193. The top staff is in treble clef, and the bottom staff is in bass clef. Measure 190 features a triplet of eighth notes (G4, A4, B4) followed by a quarter note (C5) and a quarter rest. Measure 191 features a triplet of eighth notes (C5, B4, A4) followed by a quarter note (G4) and a quarter rest. Measure 192 features a quarter note (F4), a quarter note (E4), and a quarter note (D4). Measure 193 features a quarter note (C4), a quarter note (B3), and a quarter note (A3). The bottom staff contains chords: F#4-C#5 in measure 190, G4-A4-B4 in measure 191, and F#4-C#5 in measure 192.

194

Musical score for measures 194-196. The top staff is in treble clef, and the bottom staff is in bass clef. Measure 194 features a triplet of eighth notes (G4, A4, B4) followed by a quarter note (C5) and a quarter rest. Measure 195 features a quarter note (B4), a quarter note (A4), and a quarter note (G4). Measure 196 features a quarter note (F4), a quarter note (E4), and a quarter note (D4). The bottom staff contains chords: G4-A4-B4 in measure 194, F#4-C#5 in measure 195, and F#4-C#5 in measure 196.

197

Musical score for measures 197-199. The top staff is in treble clef, and the bottom staff is in bass clef. Measure 197 features a triplet of eighth notes (G4, A4, B4) followed by a quarter note (C5) and a quarter rest. Measure 198 features a quarter note (B4), a quarter note (A4), and a quarter note (G4). Measure 199 features a quarter note (F4), a quarter note (E4), and a quarter note (D4). The bottom staff contains chords: G4-A4-B4 in measure 197, F#4-C#5 in measure 198, and F#4-C#5 in measure 199.

200

Musical score for measures 200-202. The top staff is in treble clef, and the bottom staff is in bass clef. Measure 200 features a quarter note (G4), a quarter note (A4), and a quarter note (B4). Measure 201 features a quarter note (C5), a quarter rest, and a quarter note (B4). Measure 202 features a triplet of eighth notes (G4, A4, B4) followed by a quarter note (C5) and a quarter rest. The bottom staff contains chords: G4-A4-B4 in measure 200, F#4-C#5 in measure 201, and F#4-C#5 in measure 202.

203

flute echoes (pp)

207

211

215

219

mf *p*

second sampler off granular processing cue 5

225

3

230

3

flute echoes (mf) granular processing off

235

3 *mf* *mp*

flute echoes (mf)

240

p *pp*

(ppp) ambient sound cue 3 fades

243

slowly fade away...

R. Luke DuBois

Biology I/II/III

for violin

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Musical Arts
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY
2003

© 2003
Roger Luke DuBois
All Rights Reserved

Introduction

Biology is a trio of short pieces for solo violin. It was written in the spring of 2003 for Maja Cerar. It is submitted in partial fulfillment of the requirements for the degree of Doctor of Musical Arts in the Graduate School of Arts and Sciences, Columbia University, 2003.

The duration of the three pieces should be ~1, 2, and 3 minutes, respectively.

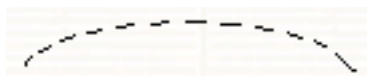
Performance Notes

The violin may be amplified, if desired. If so, an acoustic 'tone' should be maintained in the sound reinforcement.

Notation

All notation symbols in the score are standard, except where noted below.

Accidentals carry through the measure.



Indicates phrase boundaries, and should not be taken as bowing instructions or slurs.



Always as fast as possible.

Biology I

R. Luke DuBois

♩ = 90

Violin

a march for the indecisive

with growing determination...

hesitant...

confident

hesitant again.

not sure at all now.

thoughtful and drawn out

...and it's gone!

a possible solution?

arco

pizz.

arco

mp

mf

pp

pp

mf

p

p

mp

pp

ppp

rit.

mf

(at tempo)

pizz. (as fast as possible)

arco

mf

(fade to nothing)

Biology II

R. Luke DuBois

♩ = 135 like a very strange dream

Violin *arco* *pp*

9 *mp*

17 *mf*

26 *mf* *mp*

34 *mp* *sul ponticello. (let harmonics ring)*

44 *f* *p*

52 *mp* *arco* *mf* *p*

30 pensive
pp *mp*

35 *p* question?

39 answer. *mf* 5 5 more and more confident to end

42 *f* 5 5 3 3 3 3

44 *mf* *pp* molto ritard... * 3

* arpeggiate at a slower speed, but keep the arpeggios separate.