

# Programming For Music: Explorations in Abstraction



Can Ince  
School of Music, Media and Humanities  
University of Huddersfield

A thesis submitted to the University of Huddersfield in partial  
fulfilment of the requirements for the degree of  
*Master of Arts in Music*

January 2018

## Acknowledgements

I would like to thank several people who contributed to this research, either directly or indirectly. I must first thank Alexander J. Harker for his valuable supervision and mentorship throughout this research. Besides my advisor, I would like to thank to Frederic Dufeu and the rest of the Creative Coding Lab for their insightful comments and encouragement. I also thank to contributors of TidalCycles and SuperCollider for creating and actively maintaining these languages. I thank my friend Mert Toka for the stimulating discussions and his contributions on this research. Last but not the least, I would like to thank my parents and my friends for supporting me spiritually throughout this research and my life in general.

# Copyright Statement

1. The author of this thesis (including any appendices and/or schedules to this thesis) owns any copyright in it (the Copyright) and he has given The University of Huddersfield the right to use such copyright for any administrative, promotional, educational and/or teaching purposes.
2. Copies of this thesis, either in full or in extracts, may be made only in accordance with the regulations of the University Library. The details of these regulations may be obtained from the Librarian. This page must form part of any such copies made.
3. The ownership of any patents, designs, trademarks and any and all other intellectual property rights except for the Copyright (the Intellectual Property Rights) and any reproductions of copyright works, for example graphs and tables (Reproductions), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property Rights and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property Rights and/or Reproductions.

# Abstract

Creatively, algorithmic processes open up all sorts of possibilities which would be either impossible or laborious to create by hand. This thesis is my attempt to explore depth of the rhythm and time through algorithms accompanied by a software, *Siren*, which is designed for pattern sequencing.

This thesis documents an effort in attempting to develop a *novel technical approach to musical composition* that functions not just as a tool, but also as *an extended cognition that shapes the musical creative process*.

To this end, several ideas and design approaches derived from previous work in computer science, philosophy, music, and other disciplines are utilized to conceive of (and subsequently implement as a software application) a musical interface that is tailored towards algorithmic approaches to music composition. This thesis presents the result of that effort as well as the process of its creation. A discussion evaluating the abstractions and cognitive dimensions which inform the design and implementation of the application are also included.

Beside basic curiosity and experimentalism, there are several reasons why I wanted to adopt algorithmic methods and this thesis will serve as a guide and a notebook towards achieving a stability in music with fusion of various concepts.

Keywords: digital music notations, algorithmic composition, abstractions, trackers, sequencers, patterns

# Contents

<b>List of Submitted Materials</b>	<b>iii</b>
<b>List of Figures</b>	<b>iv</b>
<b>List of Tables</b>	<b>v</b>
<b>1 Context</b>	<b>1</b>
1.1 Aims . . . . .	1
1.2 Algorithms . . . . .	2
1.3 Abstractions . . . . .	4
1.4 Time . . . . .	5
1.4.1 Rhythm . . . . .	5
1.4.2 Repetition . . . . .	6
1.5 Pattern Programming . . . . .	7
1.5.1 TidalCycles . . . . .	8
1.6 Creative Coding / Live Coding . . . . .	9
1.6.1 Command-line Interface (CLI) . . . . .	10
1.7 Musical User Interfaces . . . . .	11
1.7.1 Musical Trackers . . . . .	11
<b>2 Siren: An Ecosystem for Musical Patterns</b>	<b>13</b>
2.1 An Ecosystem for Pattern Creation and Sequencing . . . . .	13
2.1.1 Hierarchical Composition . . . . .	15
2.1.2 Patterns as Functions . . . . .	16
2.1.3 Features . . . . .	16
2.1.3.1 Parameters and Modulations . . . . .	16
2.1.3.2 Mathematical Expressions . . . . .	17
2.1.3.3 Polyrhythmic Timers and Temporal Parameter . . . . .	17
2.1.3.4 Transitions . . . . .	18

2.1.3.5	Global Modifiers . . . . .	18
2.1.3.6	Pattern Roll . . . . .	18
2.2	User Interface Design Principles . . . . .	20
2.2.1	Cognitive Dimensions of Notations . . . . .	20
2.2.2	Design Heuristics for Virtuosity . . . . .	24
2.3	Siren: Implementation and Usage . . . . .	25
2.3.1	System Structure . . . . .	26
2.3.2	Modules . . . . .	27
2.3.2.1	Scenes . . . . .	27
2.3.2.2	Channels . . . . .	27
2.3.2.3	Patterns . . . . .	27
2.3.2.4	Global Modifiers . . . . .	28
2.3.2.5	Console . . . . .	29
2.3.2.6	Pattern History . . . . .	29
2.3.3	Timer Structures . . . . .	29
<b>3</b>	<b>Pieces and Music</b>	<b>32</b>
3.1	Musical Flow . . . . .	32
3.2	Commentary . . . . .	34
<b>4</b>	<b>Conclusions</b>	<b>36</b>
<b>5</b>	<b>Future Work</b>	<b>38</b>
	<b>References</b>	<b>40</b>

# List of Submitted Materials

1. Siren, An Ecosystem for Pattern Creation and Sequencing
2. Two pieces: *1sc34dl* and *foilcut*
3. Pattern studies - Recordings with Siren
4. Various custom TidalCycles functions
5. Various SuperCollider SynthDefs

# List of Figures

2.1	<i>Siren</i> , based on a hybrid visual approach that marries live-coding with a tracker-inspired environment, is meant to serve as an ecosystem for musical patterns and pattern-based compositions. . . . .	14
2.2	Example for an instance of a pattern function call. . . . .	16
2.3	Example for a pattern function located in the pattern dictionary. <i>Siren</i>	16
2.4	The pattern roll / sub-sequencer in <i>Siren</i> . . . . .	19
2.5	The block diagram for <i>Siren</i> – The “front-end” explains the hierarchical structure of <i>Siren</i> , and the communication between different components are shown with arrows. Examples and explanations of the most prominent items are presented with thin dashed boxes. Optional fields are shown with square brackets ([ ]). . . . .	26
2.6	The pattern dictionary in <i>Siren</i> . . . . .	28
2.7	Channel setup of <i>Siren</i> . . . . .	29
2.8	A channel in <i>Siren</i> . . . . .	29
2.9	The Global Modifiers module. . . . .	29
2.10	The Console module. . . . .	30
2.11	The Pattern History module. . . . .	30
2.12	An older prototype of <i>Siren</i> , with a poly-duration timer. . . . .	31
3.1	Partial visualisation of the patterns used in the <i>foilcut</i> . . . . .	35



# List of Tables

2.1	The Cognitive Dimensions from [21]. . . . .	21
-----	---	----

# Chapter 1

## Context

This chapter provides a foundation based on previous work and situates the thesis in the context of related research. To this end, six topics are discussed. The first is notion of *algorithms*, with emphasis on how they find their place in musical composition. The second is the notion of *abstractions* as they apply to the information theory, and how this theoretical concept underpins ideas that are manifest in musical interfaces in general, and *Siren* in particular. Following this is a discussion on the conceptualizations of *time* and *patterns* in music, and how different abstract understandings of time and patterns inform approaches to music composition, especially with regard to digital musical instruments. The subsequent section overviews the concept of *creative coding*, with emphasis on "live coding" of music. Finally, this chapter discusses the impact of *user interfaces* on musical creation, paying special attention to music trackers.

### 1.1 Aims

The research aims to accomplish the following:

1. Building a musical interface informed by the user interface design principles from previous research.
2. Developing an intuitive technical approach for musical composition and performance, not just as a tool, but also as an extended cognition facility that shapes the musical process.
3. Understanding design characteristics that are relevant for a software application that aims to function as an extended cognition facility, specifically for supporting a hybrid approach to live coding and composition

4. Using the aforementioned approaches in musical compositions.

## 1.2 Algorithms

The Greek philosopher, mathematician, and music theorist Pythagoras (ca. 500 B.C.) documented the relationship between music and mathematics that laid the foundation for our modern study of music theory and acoustics. The Greeks believed that the understanding of numbers was key to understanding the universe. Part of their educational system, the *quadrivium*, was based on the study of four subjects that comprised arithmetic, geometry, and astronomy, and music. Although there are numerous treatises on music theory dating from Greek antiquity, the Greeks left no explicit evidence of how mathematical procedures applied to the composition of music [53].

Since then, several categories have emerged from the study of algorithms for music composition, including the aleatoric (or chance) methods (e.g. Cage), determinacy (e.g. Schoenberg, Webern, and Berg), and stochastic (or probabilistic) methods (e.g. Xenakis and Hiller). Composers have also been applying not only mathematical models, but also biological paradigms such as L-systems [58] and genetic algorithms [22] to the creation of music. These days, since some form of almost any of the aforementioned process categories may be modeled computationally, almost any of these models may be used for music composition in computational environments [53].

Before we move on to other topics, this chapter will benefit from a more involved discussion of what an algorithm and composition is *is*.

An *algorithm* is defined as "a fixed step by step procedure for accomplishing a given result [...]" and "a defined process or set of rules that leads to and asserts development of a desired output from a given input" in the Computer Dictionary and Handbook[54]. However, the precise meaning of the term *algorithmic* often depends on the context it is used in. Several criteria have been proposed to form a generic understanding of the term. These include the requirement to have a finite number of steps, to have both well-defined input(s) to and output(s) from the algorithm, to yield a result in a finite period of time, and to have a precise definition for each step of the algorithm [53]. If the algorithm is to be programmed to function as a human being would, it can be said to involve decision-making and development procedures. In this case, the terms *random*, *stochastic*, or *aleatory* (all pertaining to chance) would more accurately describe a process, rather than *algorithmic* (i.e. using defined logical procedures) or *intelligent* (simulating human mental processes) [55]. More recently,

Autechre[56] mentioned about their use of algorithmic tools in an interview with Sound on Sound Magazine.

It seems that for a lot of people, if they hear something that doesn't sound regular, they assume it's random. If live musicians were playing it, they'd probably call it jazz or something. But the fact that it's coming out of a computer, as they perceive it, somehow seems to make it different. For me it's just messing around with a lot of analogue sequencers and drum machines. It's like saying, 'I want this to go from this beat to that beat over this amount of time, with this curve, which is shaped according to this equation.'

*Composition* is the process of creating a musical work. The meaning of the term composition relates to putting together parts into an unified whole. Moreover, the process of composing music is often characterized by trial and error. The composer executes an idea, listens, and determines if revisions are necessary. As such, the composer continuously evaluates the effectiveness of a part in relation to the whole.

In my work, I conceptualize a *compositional algorithm* as a set of rules, a description and transformations in an artificial computer language of a pattern creation process. It can be understood as a written *score* that embodies a procedure for composing. However, instead of humans instrumentalists playing the music described in the score, a computer does.

An early example of algorithmic music is the works of Iannis Xenakis. Xenakis generated his first piece *Metastaseis* (1955-56) [51] dealing with large number of data sets, and started using computers as a necessity to assist these calculations, producing three works in 1962 [61]. Since then, numerous tools have been developed for composers looking to explore such ideas. Max/Msp [50], a modulable programming environment, SuperCollider [33, 34], a platform for audio synthesis and algorithmic composition, and many others has been used by composers and researchers who have been inspired by the stochastic and dynamic algorithmic methods. Such ideas and theories can be remodeled and re-explored efficiently in these new environments and to date, there have been developments in the field through a multitude of algorithmic composition tools, which allow a composer to work more quickly by offering them a close match between the creative methodology and the algorithmic implementation. A more recent manifestation of this approach lies at the heart of the 'Algorave' movement, which is centered around live-coded music, often accompanied with live-coded visuals. The audience experience in these performances is quite rich, and accounts of

Algorave events report audiences “looking up at the projected codes rather than each other,” and that it feels “in many ways like an art performance with some dancing” [5].

## 1.3 Abstractions

The concept of ‘abstraction’ is central in computer science [57]. An ‘abstraction’ can be defined as “a concept or idea not associated with any specific instance” or “the process of formulating general concepts by abstracting common properties of instances” [43]. It is notable that the root of the word, ‘abstract’, relates to the immaterial and vague. However, abstractions in computer science don’t always have to be so, as they may represent particular domain-specific notions rather clearly.

The Free Online Dictionary of Computing defines abstraction as “[producing] a more defined version of some object by replacing variables with values (or other variables).” According to Dijkstra, abstraction in computer science is not as much about ignoring details as capturing essential details. Abstraction is the tool that gives computer science its algorithms and variables, and it permeates the whole subject [of computer science]. [13]

The instantiation of abstractions is often phrased as software reuse, as described by Krueger [29]:

Abstraction plays a central role in software reuse. Concise and expressive abstractions are essential if software artifacts are to be effectively reused. Effectiveness of a reuse technique can be evaluated in terms of cognitive distance and an intuitive gauge of the intellectual effort required to use the technique. Cognitive distance is reduced in two ways: (1) Higher level abstractions in a reuse technique reduce the effort required to go from the initial concept of a software system to representations in the reuse technique, and (2) automation reduces the effort required to go from abstractions in a reuse technique to an executable implementation.

This idea can also be applied to live coding practice, where the abstraction are essential in order to add dynamism to the sound. A written code can be propagated in different musical forms with a slight changes in the input. Here, the instantiations of abstractions may be considered as affordances for virtuosity; since a bi-directional relationship is achieved between the output realized by the system and the inputs provided by the musician (see Sec.1.6).

It's possible to realize the effectiveness of a particular method, with that method being one node in the ladder of abstraction. A stable structure for the usage of abstraction is realized with connecting the nodes. As we go higher on the abstraction level/order, the parameters that controls the abstraction is reduced. Even if the input may seem unwild, it may cause musical output to become overly dynamic.

One particular case of how such an understanding of abstractions informs the design of *Siren* is the notion that Patterns (see Section 1.5). In *Siren*, patterns are represented in a modular fashion. This modularity is supplied by the pattern functions (see Section 1.3) presented within the software, which allow for the instantiation of an abstracted patterns.

## 1.4 Time

Rhythm, meter and duration are the concepts that govern the human understanding of musical temporality. Without the differentials of rhythm and meter, it wouldn't be possible to have any distinguishing qualia with regard to the musical experience. Musical temporality is thus at the core of human existence, enabling the human consciousness to constitute meaning and to relate to being as formed.

Yet, a precise characterization of *time* and temporality as we experience it, as it relates to the evaluation of algorithms on a digital circuit, has often been purposefully ignored in computer science [38]. One of the main focus of this thesis is the integration and exploration of temporal structures in algorithmic music (see Section 2.1). To this end, different methods are used to work with time and conceptualize it. In turn, each of these different methods had a significant impact on the software and the music. Time could be understood and manipulated in many different forms and ways, some of which engender characteristics that pertain to certain musical genres or memes.

### 1.4.1 Rhythm

Rhythm is everywhere in our daily life, from heart beating to walking through brain functions. Ordinarily, we believe that our perception of it, in an abstract form, may precede the music itself. In this understanding, music is simply one of many aural phenomena made up of sound (or sounds). Sound is perceived because it is temporally formed. It is at this point that *sound* becomes *music*. Without rhythm, we would not hear sound – just as we would not recognize any letters without the understanding that such letters can be organized into *words*. In the words of Lefebvre, "everywhere where there is interaction between a place, a time and an expenditure of

energy, there is rhythm” [30]. It follows that experiential qualia can be categorized broadly into those pertaining to *repetition* (of timbre, movements, gestures, actions, situations, differences, etc.), and to *interferences*, with respect to both *linear* and cyclical processes, which can be articulated using concepts such as birth, growth, peak, decline, and end. (The third chapter of this thesis will elucidate further how this categorization pertains to my approach to an interface for musical composition and performance.)

The concepts of *repetition* and *variation* (of patterns, in time), which can manifest in balance or in a way in which one overpowers the other, supports to enable the experiences of *similarities* that arise from *repetition*, and *progression* or *generation* which can serve to both affirm or negate *anticipation* at various points in time [30]. In the musical interface that is proposed in this thesis, the articulation of such effects are supported extensively through different timer structures that allow concurrent execution of patterns with respect to different temporal intervals. Hence, it becomes possible for the composer to fluidly transition between different experiences of repetition, while retaining the pattern affordances of the *live coding* and *tracker* paradigms<sup>1</sup>.

### 1.4.2 Repetition

*Rhythm* cannot manifest without repetition. It cannot exist without reprises, and without measure. However, even in repetition, instances are not perfectly identical – absolute repetition does not exist. The relation between *repetition* and *difference* is a significant one in relation to music composition and performance. Many powerful musical forms such as the canon, the fugue, the sonata, and the rondo have been shaped around repetition as a core value [55].

One important aspect to consider when reasoning about repetition in this context is the *information differential*, i.e. the density of new information over time, and the capability of a human listener to absorb and comprehend (or apprehend) it. Much of the expressive power in musical composition relates to the manipulation of the balance between redundancy and new information. Repetition and continuity are potent tools in the composer’s arsenal, which can be utilized to manipulate the *predictability* in musical expression and to shape the ephemeral *listener’s hypothesis*. Both *cyclic repetition* and the *linear repetition* can be part of this mixture and they interfere

---

<sup>1</sup>The reader will find, in the following sections of this chapter, discussions on how significant attention to *patterns* informs the musical composition interfaces, the practice of live coding, and the affordances of musical trackers.

with one and another constantly [30]. Based on Lefebvre’s thoughts on repetition, an abstract level, almost every aspect of existence can be thought to manifest through cyclical and linear repetitions. The linear relates to the the monotony of sounds, of gestures, and of imposed structures; while cyclical repetition is periodic, restarting continuously .

In this thesis, the concept of the *meter* is considered to be one of the most important aspects of *patterns*. This is a well established phenomenon that allows us to analyze the characteristics of *rhythm* as discussed in the previous section. ”Meter is a perceptually emergent property of musical sounds, that is, an aspect of our engagement with the production and perception of tones in time” [31].

It is important to be able to distinguish between the concepts of *rhythm* and *meter*. *Rhythm* relates to the occurrence of temporal patterns that are ”phenomenally present in the music.” Such patterns can be referred to as ”rhythmic groups.” Temporal patterns, however, are not always strictly founded on the ”action duration” of musical event. A rhythmic pattern can occur ”between the the start-end points of successive events” [31]. In contrast, *meter* involves the initial perception of the listener, as well as subsequent anticipation of a series of events that the listener abstracts from the *rhythmic surface* of the music as it unfolds in time. Altering the meter of patterns manifests and manipulates *flow* in the musical structure, and thereby allows for emotion to be encoded and decomposed by the transitions and differentials between musical information.

## 1.5 Pattern Programming

Lefebvre states that music can function as “an alternative to purely mathematical models of calculation and measure” [30]. In music, by applying algebraic operations to temporal structures, such ”mathematical models of calculation and measure” can be represented as *events in time*. These temporal structures can be combined or modified to create sequential events, on a computer. Such ideas are the basis for several programming languages for the purpose of specifying musical patterns.

In SuperCollider, one of the most notable features for pattern programming is the `Pbind` class, the principal function of which is to combine various streams of information into one ‘event stream.’ Using this facility it is possible to create ‘value patterns’ which are mapped to different variables and can be used to perform in creative ways.



More recently, *Conductive* [6] has been designed as a language that offers a higher-level abstraction. In *Conductive*, for example, it is possible to generate *sets* of varying densities based on an initial pattern, and to store them in a table indexed by their level of density, so that they can be retrieved and utilized in compositions[7]. Finally, *TidalCycles* (or *Tidal*, in short) [36, 37] introduced an *embedded Domain Specific Language* (eDSL) for composing patterns as higher order structures with a highly economical syntax. The ideas encapsulated in Tidal and its implementation have greatly informed the effort that is the subject of this thesis, and warrant a more detailed discussion.

### 1.5.1 TidalCycles

In Tidal, time is conceptualized in a cyclical, rather than linear manner; and the term *arc* is used to describe a temporal range, delimited by specific begin and end time and it is based on rational subdivisions.

Tidal represents each musical event, delimited by a start and stop time that are termed the event *onset* and *offset*, as an *arc*. The association of a value pertaining to two time arcs is termed an *event*. In this case, the first arc is associated with the onset and offset of the event, and a second arc relates to its 'active' portion. This second arc is utilized in cases where an event consists of multiple peaces – when it is important for each piece to 'store' the original arc that denotes its context. Finally, in Tidal, *patterns* represent functions that associate an arc to a list of events. Patterns can be 'queried' with an arc to return a list of all events that are 'active' during a given time. These arcs may have overlapping events, which supports polyphony in music without requiring the introduction of events that deal with multiple values (even though using chords in lieu of atomic events is a possibility).

Conceptually, all Tidal patterns are infinite in length. They can cycle indefinitely, and they can be 'queried' for events at any point in time.

Tidal does not have built-in synthesis capabilities itself; rather, it is designed to be used primarily with the *SuperDirt* sampler [2], and can communicate with other instruments over the *Open Sound Control(OSC)* and *MIDI* protocols.

While it is possible to represent long-term structure in *Tidal*, the focus is on live coding situations where long-term structure is controlled and continuously manipulated by the performer [39]. This absence of a facility for defining long term structure in *Tidal* is the one of the key issues that has driven the creation of the software described in this thesis.

## 1.6 Creative Coding / Live Coding

This research has been informed by Amabile's componential theory of creativity, which rests on two important underlying assumptions [4]. First is the idea of a *continuum* where on one end is found the low, "ordinary" levels of creativity found in everyday life, and another end is characterized by the "higher" levels of creativity found in significant inventions, performances, scientific discoveries, works of art, etc. The second and related underlying assumption is that there are degrees of creativity manifest in the work of any single individual, even within one particular domain – this is what enables an individual to progress and persevere within that domain. The "level of creativity" that a person can tap into at any given point in time is a function of the *creativity components* which are in operation at that particular time, within and around the person.

In my own practice, *programming* is used consistently as a way of musical thinking. It would be fair to state that the development of a musical ecosystem involves many *creativity components* which, ultimately, enables progress within the creative process.

The word *programmer* is often used to implicitly refer to a kind of technician who tends to a computing machine. However, the same word can be used beyond this context, in relation to a *craft* which is situated in an artistic context. Confronting the singular identity of the programmer as artist is a particularly salient excerpt from John Stuart Mill [42] which was also mentioned by Donald Knuth:

Several sciences are often necessary to form the groundwork of a single art. Such is the complication of human affairs, that to enable one thing to be done, it is often requisite to know the nature and properties of many things. Art in general consists of the truths of Science, arranged in the most convenient order for practice, instead of the order which is the most convenient for thought. Science groups and arranges its truths so as to enable us to take in at one view as much as possible of the general order of the universe. Art brings together from parts of the field of science most remote from one another, the truths relating to the production of the different and heterogeneous conditions necessary to each effect which the exigencies of practical life require .

One of the key issues in the practice of creative coding is the particular amount of *time* dedicated to working on a potential implementation without yet obtaining a result. However, it can be said that there exists a potential for *enlightenment*

when working with yet-non-functional (or even malfunctioning) code. In this state, it is possible for the programmer to conceive of a potential feature where the ideas can be emergent. A decision made at this point, affects the flow of the software and eventually reflect on the musical output. Thus, in the process of developing an instrument for creative coding, one goal is to expose the programmer to such windows and perspectives. Even aspects of ordinary life can be exploited to serve the creative process in such ways, with the appropriate tools.

*Live coding* is a solid examples of *creative coding*. It is an often improvised performance practice which includes the writing and/or editing of a generative rule set that governs a *current* and ephemeral flow of artistic creation. It is an act of expression and communication that involves the creation, modification and display (as-is or as manifest in their output) of codes. Regardless of the "superficial" appearance of the live coding act, many fundamental aspects are shared with core artistic practices. The practice of live coding involves a broad range of variation and style. For example, some performances may be based on previously written code, while others may be coded from scratch. The feedback from the interpreter which can become a part of the performance – in some performances this feedback is sparse, or not shared with the audience, while in others it is verbose. Live coding is a good example to computer programming retaining aspects of a science and an art, and that the two facets complement each other beautifully[28].

While it is possible to utilize the designed software to execute a live coding performance, the main focus of this research is on the pattern creation aspect rather than performative aspects of live coding.

### 1.6.1 Command-line Interface (CLI)

The CLI was the primary means of interaction with most computer systems on computer terminals in the mid-1960s, and continued to be used as the main interaction interface throughout the 1970s and 1980s. The interface is usually implemented with a command line shell, which is a program that accepts commands as text input and converts commands into appropriate operating system functions. Meanwhile, hybrid systems for algorithmic composition that combine multiple approaches led to new possibilities for expression in live-coding [40]. These systems make use of two or more languages and communicate through various bindings like *Open Sound Control* or *Websockets*[47]. These languages are mostly domain specific hence they can be accessed directly using a CLI and the associated compiler. The principal drawback of such hybrid systems is that they may be very complicated, which proposes a high

learning curve for users and, therefore, hinders accessibility. However, hybrid systems can also offer novel "powers" to composers and performers by bringing different capabilities of different domains together.

In *Siren*, the command line mentality and the graphical ecosystem have a direct connection, where the compiled patterns are stored in a pattern history. It allows user to play along with the timers and intervene to the running patterns hence maintaining the expressivity of live coding.

## 1.7 Musical User Interfaces

The design of new interfaces, as well as the interaction between forms of music engender a compelling domain of research that links the practices of composition, performance, and improvisation in the context of the computer. Crucially, current real-time systems permit the composer to become the performer of their composition, even directly affecting the micro-structure of sound and affect in a very immediate and direct way [18]. One important consideration about this creative domain is that the *interface* shapes how the practitioner perceives and interacts with their art. In the formal systems on which this domain is founded (e.g. the implementation of computer programming languages and conventional practices of musical composition) it is precisely the design of *notation* that defines the *affordances*, i.e. what actions and expressions are possible. This notation system is provided by the programmer in the live coding (see Sec. 1.6) while in any other conventional music system it is provided by some kind of a "discrete" (abstract but always discrete) grid system.

### 1.7.1 Musical Trackers

Trackers are a class of sequencer based on a concise text notation, edited using the computer keyboard. The user interface of *Siren* is heavily influenced by the notion of a musical tracker. The tracker user interface depicts notation as rows of discrete musical events positioned in columnar channels. Each cell in a channel can hold a note, parameter change, an effect toggle and other commands. Different patterns or loops can have independent timelines, which can be organized into a sequential master-list to form a complete composition. The earliest implementations of the *tracker* concept were released for the AmigaOS platform in the late 80s and early 90s, in applications such as Ultimate Soundtracker [11] in 1987, NoiseTracker [32] in 1989, and Protracker [15] in 1990. Ultimate Soundtracker was the creation of Karsten Obarski, who built it to relieve himself of the *labor* involved in coding computer

music by hand, with the tools available to him at the time. Obarski designed a tool that graphically represents the four channels of sound on the Amiga's sound chip like a vertical piano roll. The piano roll metaphor elegantly matched the looping structure common to nearly all music playback subroutines of the SID period [14]. Trackers are concluded as the most effective digital instrument in the findings of Nash and Blackwell on *Cognitive Dimensions of Notation*[10] and will be evaluated more profoundly in the next chapter.

## Chapter 2

# Siren: An Ecosystem for Musical Patterns

This section presents *Siren*, a musical user interface envisioned as novel *ecosystem for pattern creation and sequencing*.

First, the structure of the pattern creation and sequencing ecosystem that *Siren* is based on, in terms of its data structures and underlying technologies, is described. This is followed by an introduction to the principles that informed the design and implementation of *Siren*. Subsequently, a report on the current implementation of *Siren* user interface, which is intended to function as a software application for both composition and live performance, concludes the section.

### 2.1 An Ecosystem for Pattern Creation and Sequencing

*Siren* is an ecosystem for pattern creation, live coding performance and algorithmic composition.

Here, the concept of a “pattern” is borrowed the Tidal programming language “for encoding musical patterns during improvised live coding performances” [39]. In essence, Tidal is a “pattern language” which is “embedded” in the Haskell programming language (see [23]), which offers means to represent encodings of musical patterns, a “library of pattern generators and combinators” and a scheduling system for dispatching events.

*Siren* is based on a hierarchical structure of data, and a tracker-inspired user interface, initially intending to build on the concepts and technology of Tidal. The main idea in *Siren* is to support a hybrid interaction paradigm where the musical building blocks of patterns are encoded in a textual programming language, while the

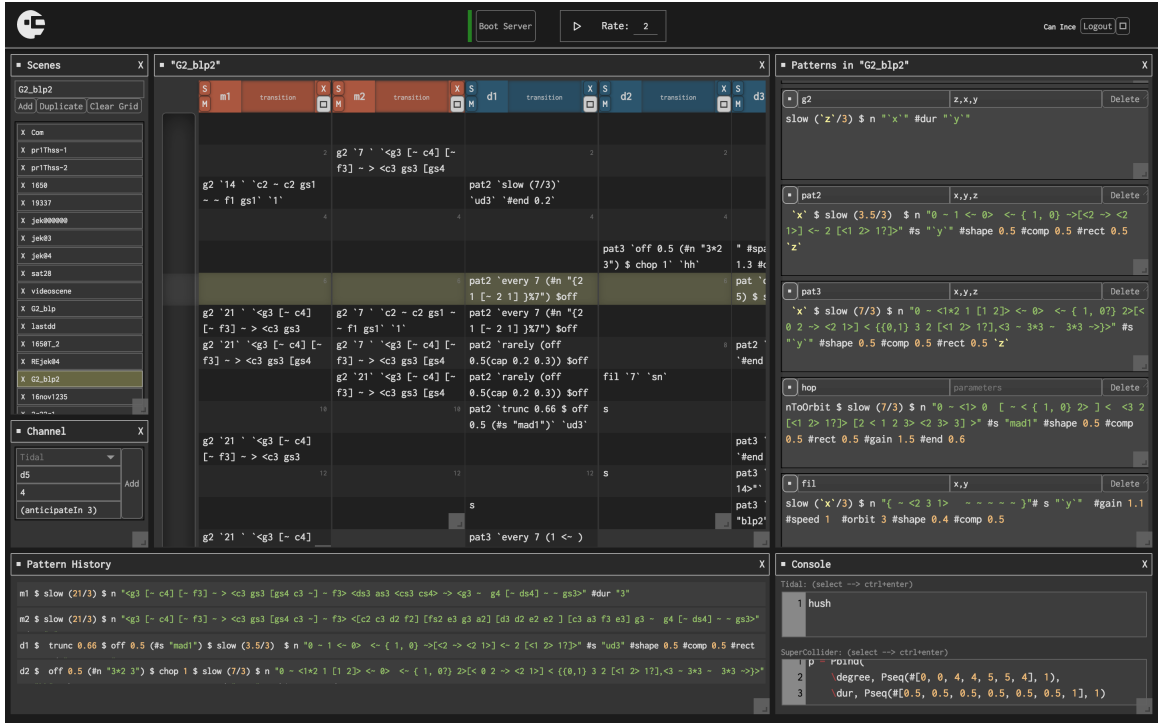


Figure 2.1: *Siren*, based on a hybrid visual approach that marries live-coding with a tracker-inspired environment, is meant to serve as an ecosystem for musical patterns and pattern-based compositions.

arranging and dispatching of patterns is done via a grid-based user interface inspired by musical trackers (see [45]). In addition to pattern arrangement, the user interface in *Siren* supports the specification of variations and transitions between patterns. Thus, it can be considered an *ecosystem* for patterns and the representation of compositions based on patterns, based on a visual approach to live performance with code, in a familiar, tracker-inspired, grid-based environment (see Fig. 2.1).

In *Siren*, patterns denote functions that are stored in a *pattern dictionary*. These pattern functions can be called from cells in the channels, and parameters that affect the behavior of the pattern functions can be specified within cells for each call. While the concepts that have inspired *Siren* are rooted in Tidal, patterns in *Siren* need not be encoded in the Tidal language. The current version of *Siren* indeed supports the use of Tidal and SuperCollider but the application of the underlying concept is not limited to these languages, and the software can be extended to use other means of encoding patterns.

In the tracker grid, per convention, each column represents a channel. Channels in *Siren* are analogous to “tracks” in contemporary digital audio workstations. In the current implementation, channels are specific to a certain pattern language—while

patterns encoded in different languages can be stored in the same pattern dictionary, each channel may only call pattern functions written in a specific language that is determined by the user upon the creation of the channel. In other words, each channel runs an interpreter or compiler for a specific programming language as its back-end. Each cell in a channel can be filled with a single *function call*, the definition of which is found in the pattern dictionary. As such, cells in channels contain only the “call” for a function, which, as is conventional in many programming languages, comprises its name and the parameters to be passed to it.

A function that is defined in the pattern dictionary can be called from any cell of any channel (provided it is written in a language that the channel’s interpreter can execute). The execution happens thusly: A timer, or scheduling system, in *Siren* scans each channel from top to bottom, triggering pattern functions in cells as they are encountered. When a cell is triggered by the timer, *Siren* performs a look-up in the pattern dictionary. Once the desired pattern is found in the dictionary, the pattern is called by parsing its parameters and replacing them with the user input provided in the calling cell. If the related pattern in the dictionary is reconstructed correctly, the channel’s interpreter or compiler executes the function. For example, in the case of Tidal channels, the Glasgow Haskell Compiler (GHC) [26] is used to compile patterns to be parsed by Tidal, which can then be sent as an OSC [60, 59] message to the SuperCollider software (which runs separately from *Siren*). For SuperCollider channels, SCLang [52] is used for the communication between the interface and SCsynth.

### 2.1.1 Hierarchical Composition

The main musical structure in *Siren* is the concept of the *scene* (see Sec. 2.3.2.1), which acts as a top-level container and a framework for a composition. Each scene could be thought of as a grid where each column is a ‘channel’ and each row denotes time steps. A timer cycles through the rows, from top to bottom, and triggers the content of each cell. Each scene has a *pattern dictionary* (Fig. 2.1) for the storage of *pattern function*, their parameters and implementations (Fig. 2.6). Instances of these pattern functions (Fig. 2.3) can be written into the grid to act as a function call to patterns on trigger. The size of the grid is bound to the number of channels, in other words, the number of columns in the grid. A user can create multiple scenes, each of which has unique pattern functions and channels. A *scene* can be used for sketchbook or while another one can represent a composition.



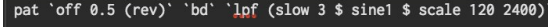


Figure 2.2: Example for an instance of a pattern function call.

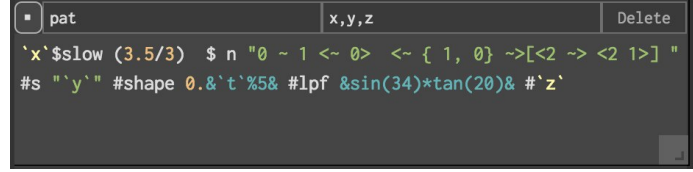


Figure 2.3: Example for a pattern function located in the pattern dictionary. *Siren*

## 2.1.2 Patterns as Functions

The system is designed to treat stored patterns as functions. Patterns are referenced as *function calls* (e.g. Fig. 2.2), which is written in one of the cells in the channel, and the *pattern function* (e.g. Fig. 2.3), located in the *pattern dictionary*. The *Pattern function* contains the optional parameters and the implementation of the instance. Furthermore, instance has two components, function name and parameters that are enclosed in grave accents in turn ( ` ).

## 2.1.3 Features

Some of the most powerful aspects of *Siren* are the features that are added on top of pattern manipulation mechanisms.

### 2.1.3.1 Parameters and Modulations

The pattern functions on *Siren* can contain any number of *literal* or *random* parameters.

A *literal* parameter is defined as any substring enclosed in grave accents (except for ‘`t`’, which is reserved for the *temporal* parameter (see Sec. 2.1.3.3)). They can be used in multiple places in the same instance, resulting in the ability to create complex relationships and modulations, especially when used with mathematical expressions (see Sec. 2.1.3.2). These parameters do not have a specific type, and any kind of input that reconstructs a syntactically correct pattern is accepted.

*Siren* also introduces *random* parameters, which can be constrained within given boundaries (e.g. `|0,3|` ). This adds an extra element of randomness to the compositions. This aspect of *Siren* is particularly interesting when used to create multi-level

randomness inside a pattern (e.g. `irand('x')` where  $x$  is the value of the random parameter). Or in a more complex case,

```
d1 $ sound "foil0 gen0" #speed (scale 0.5 'x' $ rand)
```

It is possible to change boundaries of randomness in the speed parameter with generating another bounded random parameter.

### 2.1.3.2 Mathematical Expressions

The system allows use of a wide range of mathematical expressions that enhance the computational and algorithmic aspect of pattern creation. These expressions can be employed in any part of the implementation of pattern functions, and upon successful evaluation, the expression is replaced with the final value. These expressions are notated by surrounding the expression with ampersands (&). Integration with parameters, especially the constantly incremented temporal parameter (i.e. `'t'`), opens up new relationships for certain parts of the pattern, such as:

```
slow &log('t', 2)& $ sound "gen0" # delay "0.'t'"
```

Here, the duration of triggers is in relation with the logarithm of the speed of the pattern. The expressions support numerical spaces, symbolic calculations, trigonometry, vector and matrix arithmetic [27].

### 2.1.3.3 Polyrhythmic Timers and Temporal Parameter

Polyrhythmic events are possible through the implementation of dynamic channels. Channels can have different numbers of steps, and by incrementing the number of steps, events can be quickly created or extended. The tracker's timer is represented with a temporal parameter (`'t'`), which provides the current time to *pattern functions* at the moment of execution (see Fig. 2.3).

```
d1 $ sound "gen0" #speed (scale 0.'t' 'x' $ sine)
```

In this case, the lower boundary of the randomness is determined by the timer's position in the channel.

#### 2.1.3.4 Transitions

In Tidal, the functionality to specify transitions between patterns is provided. However, the transitions must be specified per pattern. *Siren* provides the feature to specify a Tidal transition function for each channel only once, and have the transition function affect all of the patterns in the channel.

```
t1 (clutchIn 4) $ sound "gen1"
```

Alternatively, a Tidal pattern can be used as a single shot by utilizing the transition function named **mortal** which degrades the new pattern over the given time. This allows triggering single shots and short patterns which can be used as filler or transitive elements which is commonly used in the conventional trackers (see Sec. 1.7.1).

#### 2.1.3.5 Global Modifiers

Another feature of *Siren* is that there are *global transformations* and *parameters* which act on all of the channels, by either prepending transformation functions or appending parameters to the patterns. Some examples of parameters, **#speed -1** reverses current playback on all channels, **#coarse 2** halves the sampling rate of the selected channels, and so on. Within this module, there is also a sequencer which randomly selects saved global parameters and applies them to the channels with different time intervals. This module is designed for dramatic changes in the musical output as the sequencer operates on its own temporal structure, it accepts four different time intervals and applies a randomly selected function pairs to channels and then waits for the given interval.

#### 2.1.3.6 Pattern Roll

It's possible for algorithms to create patterns where it could be too laborious to create by hand. However, while these patterns can be interesting, they sometimes may lack the precision required in a composition. In theory, it's possible to "tune" a pattern to serve the desired purpose but it may require a profound change in the very core of the algorithm; yet it is probable that it may not be as precise as it's desired to be. While in step sequencers, this precision is supplied by its step-wise nature where each step is executed individually.

*Pattern roll*, also known as *canvas* or *sub-sequencer* (Fig. 2.4), is the second sequencer of *Siren* and can be accessed from the modules menu (see Sec. 2.3.2). This

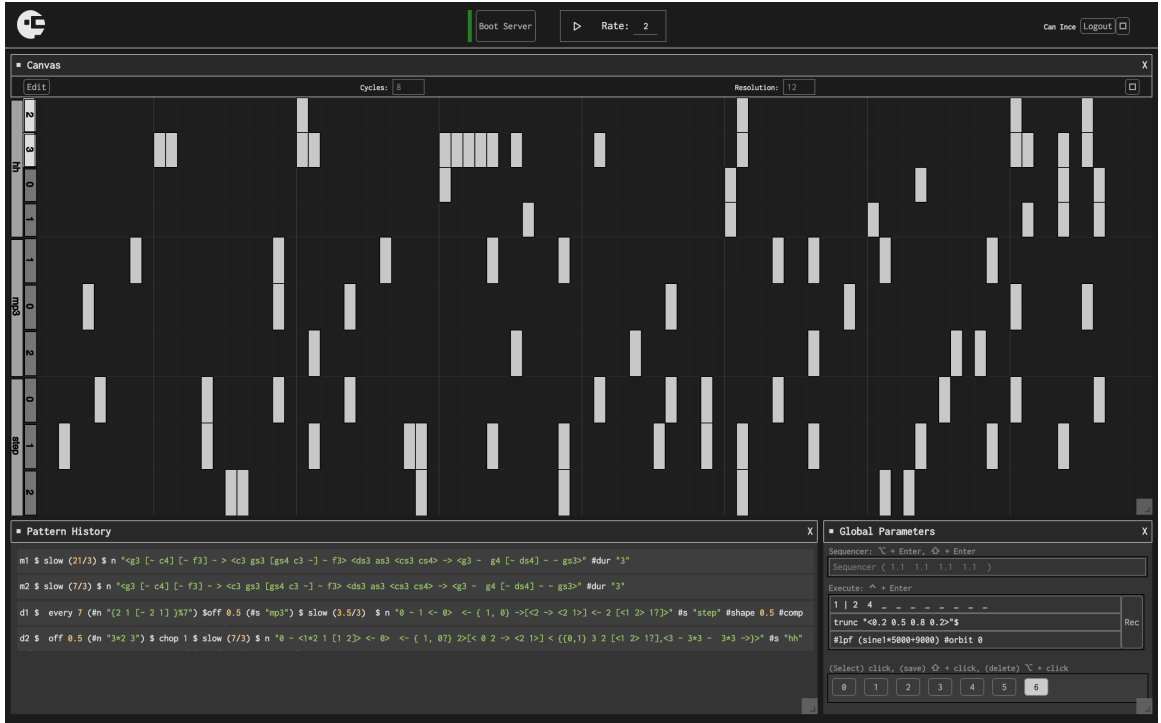


Figure 2.4: The pattern roll / sub-sequencer in *Siren*.

module is dedicated to recording instances of *SuperDirt* playback and it serves as a visual tool to understand relationships between individual triggers. Algorithmic patterns can then be modified using this piano-roll inspired module. This allows specific refinements to the patterns where it could be too difficult establish the desired effect with modifying the original pattern function.

The working principle of this sequencer is similar to the conventional step sequencers. It records the running patterns by listening for the playback messages from SuperCollider OSC function. Upon parsing the messages, note and/or sample numbers are placed in the data structure in which the vertical dimension lists unique samples or notes, and the horizontal dimension denotes the time.

This module supports various edit operations to the recorded sequence, for example, adding, deleting, or otherwise modifying notes. Each node in the timeline consists of parameters emitted from SuperCollider (e.g. **note**, **speed**, **coarse**, etc). It is possible to change the length of a sequence, as well as the tempo. The tempo of the sequence can be synced to Tidal or TidaLink (a wrapper for Link connection with Ableton Live and other Link compatible software) but it can also operate within its own domain.

## 2.2 User Interface Design Principles

A number of design principles have guided the development of *Siren*. Its user interface has been inspired by the previous works exploring the psychology of interacting with notations in the field of programming. These works break different factors of the software designer’s user experience into cognitive dimensions that help to paint a broad picture of the user experience involved with editing code and crafting software systems.

The first strand of ideas to inform the effort is the *Cognitive Dimensions of Notations* framework, originally introduced by Green for the purposes of investigating the psychology of programming languages [20], and subsequently adapted towards music notation systems and musical user interfaces [10, 9, 46, 44]. The second is the *Design Heuristics for Virtuosity* proposed by Nash and Blackwell [12], which builds on the cognitive dimensions framework and focuses on attributes that support virtuosity in musical creation interfaces.

### 2.2.1 Cognitive Dimensions of Notations

In any attempt to interact with a musical system, there will be many inescapable limitations. The interaction is always mediated through an abstract layer of notation, e.g. a stave, sequencer or waveform metaphor. The process of *sketching* illustrates how experimenting with notations can be used to support creativity. This encourages greater optimism about the opportunities afforded by notation-mediated music interaction (see [10]). In the words of Nash and Blackwell [46]:

Notations, and the interfaces used to edit them, may provide a description of end product (be informative), define an exact specification of it (be significant), have editing actions offer rapid feedback (be responsive), or be inseparably and continuously coupled to the product itself (be live). This theory demonstrates how non-realtime, notation-mediated interaction can support focused, immersive, energetic, and intrinsically rewarding musical experiences, and to what extent they are supported in the interfaces of music production software. Users are shown to maintain liveness through a rapid, iterative edit cycle that integrates audio and visual feedback.

Based on the above, a number of *cognitive dimensions* are summarized below, based on the research in cognitive science, but shaped to serve as a practical analysis and guiding tool for interaction designers, researchers, and programming language

Dimension	Definition
Abstraction	Types and availability of abstraction mechanisms
Hidden dependencies	Important links between entities are not visible
Premature commitment	Constraints on the order of doing things
Secondary notation	Extra information in means other than formal syntax
Viscosity	Resistance to change
Visibility	Ability to view components easily
Closeness	Mapping closeness of representation to domain
Consistency	Similar semantics are expressed in similar syntactic forms
Diffuseness	Verbosity of language
Error-proneness	Notation invites mistakes
Hard mental operations	High demand on cognitive resources
Progressive evaluation	Work-to-date can be checked at any time
Provisionality	Degree of commitment to actions or marks
Role-expressiveness	The purpose of a component is readily inferred

Table 2.1: The Cognitive Dimensions from [21].

architects. “Each dimension is intended to describe a distinct factor related to the usability of a particular notation. The goal is for each to relate to properties such as *granularity* (considered on a continuous scale from high to low), *orthogonality* (independence from other dimensions), *polarity* (not necessarily in terms of the ‘good’ and ‘bad,’ but characterizing ‘desirability’ in a continuous manner, for a given context), and *applicability* (in terms of a broad relevance to any kind of notation)” [44]. From Green and Blackwell [21], these dimensions, along with their definitions, are listed in Table 2.1.

Building on these dimensions, Nash formulates a set of questions to guide user interface designs in terms of more concrete design considerations [44]:

1. “How easy is it to view and find elements or parts of the music during editing?”
2. “How easy is it to compare elements within the music?”
3. “How explicit are the relationships between related elements in the notation?”
4. “When writing music, are there difficult things to work out in your head?”
5. “How easy is it to stop and check your progress during editing?”
6. “How concise is the notation? What is the balance between detail and overview?”

7. “Is it possible to sketch things out and play with ideas without being too precise about the exact result?”
8. “How easy is it to make informal notes to capture ideas outside the formal rules of the notation?”
9. “Where aspects of the notation mean similar things, is the similarity clear in the way they appear?”
10. “Is it easy to go back and make changes to the music?”
11. “Is it easy to see what each part is for, in the overall format of the notation?”
12. “Do edits have to be performed in a prescribed order, requiring you to plan or think ahead?”
13. “How easy is it to make annoying mistakes?”
14. “Does the notation match how you describe the music yourself?”
15. “How can the notation be customized, adapted, or used beyond its intended use?”
16. “How easy is it to master the notation? Where is the respective threshold for novices and ceiling for experts?”

These abstract cognitive dimensions and higher-level guiding questions have been one component of the principles on which the design of the interface is based. It would not be tractable to formulate one-to-one mappings between each feature or design element in *Siren* and a specific cognitive dimension or guiding question, however, below I can offer some comments on how exactly these concepts manifest in the final design. More detail on the final visual and interaction design and implementation, as well as how various features relate to the considerations enumerated above will be the subject of the next section.

In *Siren*, the approach of the “pattern dictionary” (Fig. 2.6) plays a critical role in editing musical compositions. In conventional step sequencers, step editing needs to be done by directly accessing its dedicated *menu* or similarly referenced location within the system. As opposed to that, in *Siren* there are no constraints as to where the step-wise edit options needs be performed. This makes it easy to immediately find musical elements and parts, which are already laid out in full view.

*Siren* strives to provide multiple ways for accomplishing the same musical result using different structures and utilizing different pattern languages (some are more accurate and more “loyal” than other with respect to an initial idea). *Global modifiers* allows direct modifications to the running patterns which could be seen as a tie with the notions of diffuseness and role expressiveness (Table 2.1).

The system allows musical ideas to be componentized and re-used with its hierarchical structure (see Sec. 2.1.1), and the user interface provides methods for this without adding further distractions to realizing musical ideas. The modularity of musical expressions in *Siren* is an innate feature of the user interface paradigm and layout system.

*Siren* leverages different abstractions in order to make good use of the visual space. The essence of the notation is initially determined in the pattern language, and its progression corresponds with a secondary timer structure. In terms of progression, these abstractions provides a great amount of freedom, as it’s possible to apply modulations to the core parts of the patterns.

Sketching is one of the salient aspects of live coding practice – the code written for a live performance could end up as a sketch after the performance, if not deleted, and serve as the basis for a later composition. When beginning a composition, there is no idea, and hence, no exactness. Findings in the initial experimentation stage lay out the fundamental ideas, which are subsequently implemented in a more exacting form in order to create a composition. In *Siren*, every scene can be utilized as a sketchbook to incubate compositional ideas. As such, an initial sketch can be created very quickly by either utilizing the pattern language (on the console) and using a few channels. From this view, the compositional environment in *Siren* enables low-viscosity workflows and secondary notations.

*Global modifiers* (Fig. 2.9) in *Siren* afford global modulations to the all active patterns. Using this feature, implementing any ‘informal’ decision which would affect the selected patterns is trivial. As such, this enables modulations to parameter values to be implemented using a selection of abstractions, without much premature commitment, while enabling provisionality.

Since *Siren* is, in its essence, a *layer of abstraction* on text-based programming environments, it is often very easy to go back and make changes to any scene or pattern within seconds which would affect the musical output.

In terms of the notations utilized for each pattern, it is very easy to differentiate the relationship between other patterns by their names and channel placements



(one possible limitation is that, within a channel, it may be difficult to identify two instances of the same pattern that uses different parameters).

Edits can be performed in any order or any part of the composition can be theoretically created without listening to it. This ability allows for musical accidents, which may ultimately turn out to be sonically enjoyable. On the other hand, in conventional step sequencers, the edits need to be planned if the aim is to introduce, for instance, some big breakdown in the middle of the composition – this requires precise adjustments before and after the event. Comparatively, the scene concept in *Siren* can be utilized to introduce a break or a verse. In a way, it is designed to allow mistakes which may end up fueling inspiration.

Some limitations of the implementation include the absence of micro-tonality and the lack of extensive customizability for the notation. In *Siren*, currently, the only form of notation that can be used to express fine-grained ideas is by writing code. The upside to this is that code can allow for highly sophisticated creative possibilities.

### 2.2.2 Design Heuristics for Virtuosity

The design of *Siren* is further informed by the Nash and Blackwell’s *Design Heuristics for Virtuosity* [12], which has the “cognitive dimensions” framework at its foundation, and focuses on foregrounding factors that entail “virtuosity” in the user interfaces for live musical performance. These heuristics are:

1. Support learning, memorization and prediction (“recall rather than recognition”)
2. Support rapid feedback cycles and responsiveness
3. Minimize domain abstractions and metaphors
4. Support consistent output and focused, modeless input
5. Support informal interaction and secondary notation

These heuristics have been chosen due to the notion that rather than more conventional approaches to usability and user experience design in human-computer interaction, user interface designs for supporting musical creativity are better informed by considering issues such as *flow*, *liveness*, and *virtuosity* that are of importance in musical contexts [45, 46, 12].

In *Siren*, specifically, the principle of “recall rather than recognition (1)” and support for “rapid feedback cycles and responsiveness (2)” have been considered as guiding notions for the design of the user interface, its underlying concepts, and its implementation. Moreover, “support for informal interaction and secondary notation (5)” is allowed through a fusion of *textual* and *visual* approaches to programming and composition. The fusion of these approaches is also visible in consideration of the *dimensionality* of representations in terms of both their visual manifestation and the mental models they encourage.

*Textual* programming can be considered one-dimensional because the relationships are encoded in text, and thus, are represented based on adjacency. In visual programming (for example, in *Max/MSP* [17] or *PD* [49]), putting objects on top of each other is possible and the program still runs. However, since the encoding is compiled into machine instructions regardless of its visual representation or its user interface, this dimensionality only pertains to the experience of the user, not to the workings of the process itself [41]. In *textual programming* however, these visual manifestations also represent the workings of the compiler. In case of *Siren*, the introduction of a temporal structure, which is essentially textual but communicated as a visual component, eases the cognitive load on the user. Furthermore, in *Siren*, the notion of supporting “consistent output and focused, modeless input (4)” is addressed through the use of a single view that exposes almost all of the elements of a composition.

While these four heuristics (1, 2, 4, 5) are well-supported in *Siren*, a concession that has to be made is that in terms of “minimizing domain abstractions and metaphors (3)”, the design of *Siren* falls short in that it relies on an integration of two separate domains of musical practice: live-coding and tracker-based composition. However, still, *Siren* observes this heuristic by striving to follow already established abstractions and metaphors. Therefore, the design ensures that the users who are readily familiar with the live-coding and tracker environments can intuitively start using the main concepts of the system without intricate knowledge about the interface.

## 2.3 Siren: Implementation and Usage

*Siren* is a JavaScript-based application. The back-end, which interfaces with GHC<sup>1</sup> and SuperCollider, is built using *Node.js* [16]. *React.js* library [24] was chosen to build the user interface, due to its stability and active user community.

---

<sup>1</sup>Glasgow Haskell Compiler



JavaScript library for communicating with and controlling SuperCollider. The back-end starts a terminal that communicates directly with the compiler, and compiles the given Haskell code in the same way as contemporary text editors such as Atom, Emacs and Vim do.

## 2.3.2 Modules

In *Siren*, it's possible for a user to save four custom layouts with different modules. For example, one layout can maximize the channels to take up the full screen, allowing the user to focus on sequencing, or the console could be made to take up the full screen for a user experience similar to a text editor, favored by programmers. These layouts can be saved within the right-click menu. Navigating between the layouts is also possible using convenient key-bindings.

### 2.3.2.1 Scenes

*Scenes* are the essential component of *Siren*. A scene serves as a container mechanism for patterns and channels. A scene can be added, duplicated or deleted.

### 2.3.2.2 Channels

Channels can be added using the 'Channel' module and consists of 'type', 'name', 'step' and an optional 'transition' parameters for initiation. Once a channel is added to the sequencer, the parameters and layout can be adjusted dynamically. Patterns can be looked up from the dictionary with their names and parameters. When a cell is active, it triggers the pattern with appropriate name and applies parameters in an ordered fashion. (see Sec. 2.1.3.1)

### 2.3.2.3 Patterns

Tidal patterns are stored in the 'dictionary'. This dictionary is unique for each scene and interacts with the sequencer in terms of parameters and calls.

The syntax to be used for encoding patterns in each entry in the pattern dictionary is determined by the channel definition, which determines the language in which the pattern will be written. One additional feature that is specific to *Siren* is that parameters can be provided instead of constants (see Sec. 2.1.3.1), and specifically when Tidal is used as the pattern language, the channel names must be omitted. Different types of parameters are accepted within the patterns as explained in Section 2.1.3.1.



Figure 2.6: The pattern dictionary in *Siren*.

### 2.3.2.4 Global Modifiers

The Global Modifiers module comprises 3 functional sections: an *Execute* section where the main functionality of the module is controlled, a *sequencer* section to modify temporal flow and *the memory banks* below which allow saved modifiers to be recalled. In the *Execute* section there are two sections dedicated to appending and prepending to the running patterns. ‘*ctrl+enter*’ applies the *modifiers* to the patterns. These sections can be saved and recalled by creating presets. ‘*shift+click*’ clears the desired slot and ‘*alt+click*’ overwrites it. Pressing ‘*Rec*’ button saves the active modifiers. These modifiers are applied to the patterns shown in the pattern history section (i.e. active patterns). Channels that you want to modify can also be specified using the ‘channel’ section in the sub-menu. Writing ‘1 2’ will make the modifiers only affect the first two channels, ‘0’ is a special case and means that modifiers will be applied to all channels in the scene. The third section is dedicated to a sequencer to be able to sequence saved *global modifiers* by specifying the time intervals of between activation. One caveat regarding this module is that the design lacks visual clues that would better communicate its functionality, which will be implemented in future work..

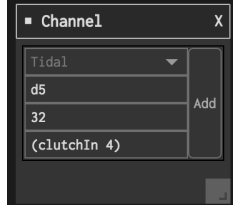


Figure 2.7: Channel setup of *Siren*.

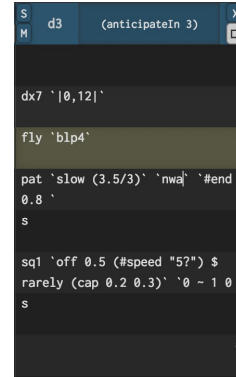


Figure 2.8: A channel in *Siren*.

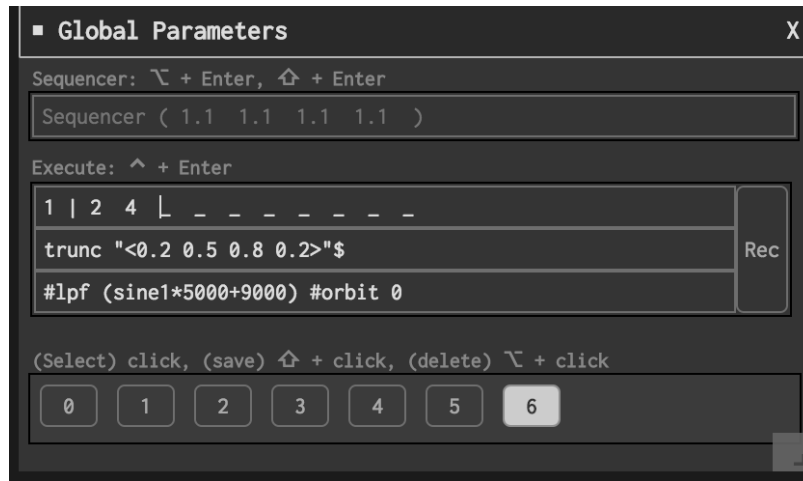


Figure 2.9: The Global Modifiers module.

### 2.3.2.5 Console

This module serves as a *CLI* (Command-Line Interface)(see Sec. 1.6.1) to GHC and SuperCollider. The compiled patterns can be monitored in the *Pattern History* module (see Sec. 2.3.2.6).

### 2.3.2.6 Pattern History

This module stores successfully compiled patterns and keeps track of the running sequences. In the back-end, this module communicates with the *Global Modifiers* (see Sec. 2.3.2.4) module to affect the running patterns.

## 2.3.3 Timer Structures

The creation of a pattern in *Siren* introduces a rhythmic element (see Sec. 1.4.1) or a cycle within its temporal bounds and linearity. Sequencing the code allows the



Figure 2.10: The Console module.

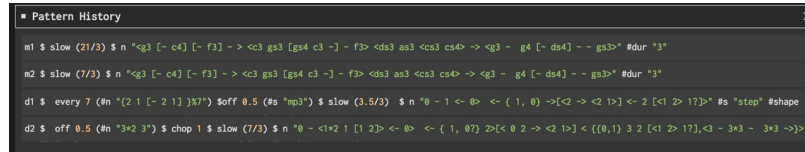


Figure 2.11: The Pattern History module.

performer/composer to break out of linearity by modifying the cycle with different variables and transitions while allowing events remain coherent in their single and linear clock cycles. This mechanism can be exploited to expand a sense of repetition (see Sec. 1.4.2) and structure on a larger scale.

Throughout the development process of *Siren*, experiments have been conducted by implementing various timer structures. An early version, for example, had been designed with unique timer controls for each channel. Although such a design allowed for a great amount of freedom, the approach was not geared towards composition but more orientated towards performative use of the software. Such an implementation also led to technical challenges as the timers were operating on their own ‘clock’, hence having no synchronization with the underlying pattern mechanism.

The current version of *Siren* implements a timer based on a concurrent tick mechanism, generated from Tidal or *Link* [1]. This allows events to be compiled at start of time-stamped bars and provides a more precise timing mechanism.

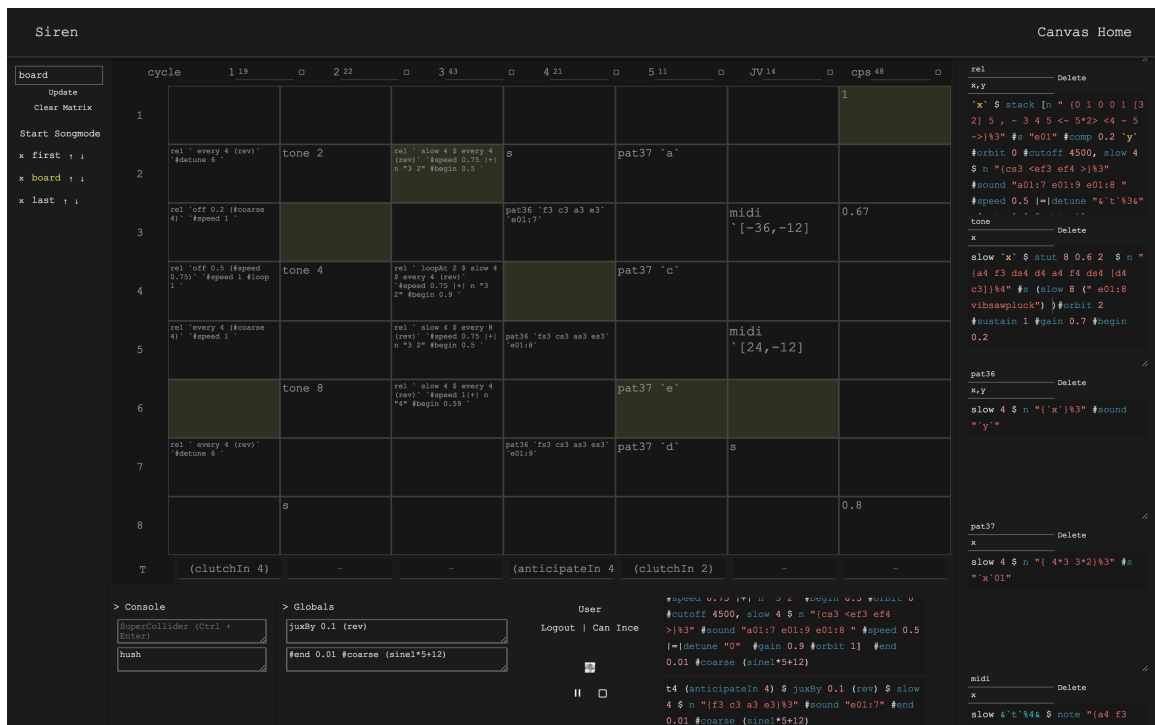


Figure 2.12: An older prototype of *Siren*, with a poly-duration timer.



# Chapter 3

## Pieces and Music

In both the creation of *Siren* and the creation of music using *Siren*, I worked towards creating structures in novel experimental forms. As I was building *Siren*, my aim was to focus on automation and abstractions (see Sec. 1.3). There are many different practices on how to approach abstractions when creating a piece or in a live performance. How much of the software or the patterns should be represented in an higher level of abstraction? How precise should the editing be?

Following a spirit of experimentation centered around answering these questions, part science and part art, in which I hypothesize on chains of digital signal processing and patterns that are possible to create with my set of tools. Then, I test my hypotheses to see if they work, and where they lead creatively. My efforts in the software design has led me to try many different methods, both technically and musically (see Sec. 2.2). My argument is that the method one chooses uses in his/her creative process shapes the musical output, as it propagates in a path which is shaped partly by imagination and by a different way of hearing, creating and experiencing the sounds of the world. Moreover, my composition practice is self-taught over several years of working with different sequencers and systems, fiddling with the menus and notations, and analyzing and understanding their relations.

In this chapter, I will compare and analyze the methods I used in musical creation, as well as examining the underlying pattern structures and features of *Siren* that influenced the musical compositions. The mentioned techniques below can be heard in example in the audio files provided under the "Pattern Studies" folder.

### 3.1 Musical Flow

In this section, I will be discussing various aspects of the patterns and my approach on them.

In *Siren*, the main focus is on spot-on debugging, in reference to just-in-time (or edit-and-continue) debugging where a running program can be stopped and edited, and can then continue execution without needing to restart. The frequency with which I move between editing the pattern, listening to the output and filling up the steps with variations of patterns has become a well-learned, reflexive motor sequence, almost an instinctive response to the creation of a new material. Playback, while manually triggered, thus becomes closely coupled with editing, enables a form of *liveness* in my process. This iterative technique enables me to quickly *sketch* and experiment with different ideas. I approached patterns with an aim to use them with different transformers, and seeing how flexible I could make them in terms of being able to reuse them in different contexts. I concentrated on melody, harmony counterpoint with the use of atonal percussive elements. It's usually a conversation with the system before settling down to a structure.

1. Program the primary patterns with using the Command-Line Interface.
2. Define a number of channels and pattern functions in the dictionary to be used in the composition.
3. Compose patterns within the channels by modulating parameters their parameters. At each step, judge the quality of the compiled pattern.
4. Create several sections within a scene or inherit another scene from the current one.
5. Improvise with the global modifiers while recording the scene.

*Syncopation* is a key aspect in the rhythmic patterns employed in my compositions. Syncopation in Tidal patterns can be achieved using subdivisions. Variations of these subdivisions create a dynamism on the rhythmic perception of patterns. However, these variations may still sound static without any *unanticipated events*, i.e. some level of randomness. This brings up the question of the extent to which the repetition in music must be consciously perceivable for it to provide the desired experience. This feeling can be modulated by using step-wise modulations with *Siren* timer. I've used algorithmic transformations extensively to redefine rhythmic structure of the patterns to achieve various musical forms and variations. The most notable modulations in this case being changes of the rhythmic density and sound sources. Being able to modulate a programming language unquestionably expanded my creative process by lifting the limitations on modulations and progressions. One of

the most interesting example is to be able to modulate the transformers that operates within the same bounds.

```
d1 $ 'x' 0.5 $ n "0 1" #s "gen0"
```

In the pattern above, the `x` parameter can be instantiated with different transformers such as *off* or *slow*, allowing modulations to create phrases and variations.

Another method that I've found effective is sound source modulation. Combining this with *Siren*'s features can result in complex textural development but it also may cause a distraction as the number of samples in the pool increases. I found myself suffering from that and wrote a script to set the samples with appropriate indexing and folder structure. As an example, the folder which contained the "snare" was tagged with an index of "1" and so on. The generated folders were named and indexed as *gen(i)*

```
d1 $ sound "gen:'x'"
```

In this case, it's possible to change the sound source by applying different integers to the parameter `x` (see Sec. 2.1.3.1) to target different sound sources. While this method has been useful for the initial phases, later I adopted to different procedures and approaches on the samples such as synthesis and field recordings.

While these can be seen as the tools and tricks, I also approached this process with an aim to map various philosophical thoughts on *linear* and *cyclic* time relations to the *musical patterns*. Both *metric* changes and unanticipated transitions of patterns establish a balance between the linear and cyclic (see Sec. 1.4). *Polyrhythms* and otherwise ambiguous rhythms can thus be seen as presenting to the listener a bistable percept [48] that affords rhythmic tension and embodied engagement. The  $N$  cycle is a powerful constraint on metric structure and complexity of Tidal patterns. It's possible to represent of the same patterns in different ways without specifically considering its metric structure. In my case, I always focused on building the metric structure of a pattern to fit with different metric forms which may be introduced later in the composition.

## 3.2 Commentary

In the track named *1sc34dl*: I tried to map cyclical and linear time relations to the interactions of the sounds. In this composition, I aimed to evoke the feeling of a certain event being pushed further as the composition progresses, while other prior

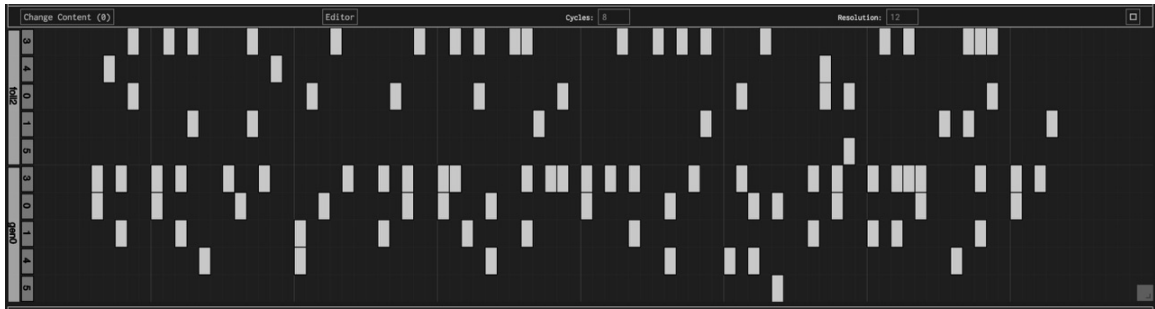


Figure 3.1: Partial visualisation of the patterns used in the *foilcut*

events are being mashed together. It was generated by utilizing a several channels with small number of steps. The *global sequencer* was set to *silence* the patterns after the specified intervals.

In the track named *foilcut*: I aimed to create a sense of repetition with the use of atonal percussive sounds, replacing snare sounds with sounds of the ice sheets being hit by small stones and sound of claps echoing off a large room. The idea is for the dense rhythmic triplet pattern to run under a noisy but melodic material. Introduced materials are rhythmically and harmonically in contrast with the main pattern hence functions as a filler in relation to it. The crunchy bass determines the sections in the composition while momentarily changing the overall timbre and degrading the percussive patterns. The relation between meters of the patterns is the main aspect that determines the overall groove of the composition. Noisy hats and FM sounds runs on four quarter-notes as opposed to main pattern's *meter* which is being modulated between three and five quarter-notes. There are also effects *comb* filters and waveshapers being applied to patterns by using the global modifiers(See Fig. 2.9). Going back and forth with these routines, it's possible to create a composition while also running the timer of the tracker interface. Even though channels in *Siren* can be set to stop once they reach the last step, I often found myself *sculpting algorithms* until I degrade them over time. I work from a bank of materials in a variety of combinations. To some degree, most of the output actually resembles a codified track which is being split up and re-constructed in the each session.

# Chapter 4

## Conclusions

This thesis described efforts to design a musical user interface envisioned as novel *ecosystem for pattern creation and sequencing*. Throughout this journey, I’ve encountered various ideas and by analyzing some of them, I was able to make conscious decisions how I would like to utilize programming in music. I also noticed that

Even though it is still at a highly experimental stage, *Siren* has gained significant attention from the live-coding community. I have been invited to perform at Algorave’s “5th birthday online live stream” [3], and *Siren*. It has been subsequently featured in the TOPLAP website, “the home of live coding” [35]. For the academic community I have published a conference paper at the 2017 International Computer Music Conference (ICMC) titled *Siren: Hierarchical Composition Interface* [25]. In addition to the conference proceedings, *Siren* has been featured in a recently crowd-sourced book on electronic music instruments, *Push Turn Move* [8], beside its predecessors such as *SuperCollider*, *PureData*, *TidalCycles*, and *SonicPI*. I have also recently performed with *Siren* at the Pattern Studies Radio<sup>1</sup>.

From the perspective of my personal musical and creative journey, after developing *Siren*, I can’t bring myself to use a traditional DAW for much more than mastering and mixing anymore. The way I work with *Siren* feels much more efficient. I get very excited about the idea of opening up programmatic descriptions of musical data because it opens up so many opportunities for complexity and variation that are difficult to explore otherwise, especially since this is something I’ve struggled with ever since I started making music. I believe strongly that computer music is so interesting because it removes so many limitations in regards to how a human can create and organize sounds. But I also acknowledge that I’m the most productive and creative when I’m working within a system with very well defined boundaries

---

<sup>1</sup>patternstudiesradio.com

and limitations. Artists imposing arbitrary limitations on themselves to encourage creativity is a common practice in a slew of creative domains. There is probably a middle ground here, where the most productive path to this flavor of creativity is the artists empowering themselves with the knowledge and tools to imagine and create their own closed systems for musical/artistic experimentation. As such, in *Siren*, the aim has been to achieve a balance in the design of a musical environment in terms of constraints and possibilities. From my own experience as a composer and performer using the software, I have been able to see that this has been achieved.

# Chapter 5

## Future Work

There are a number of features planned to be added on top of the current structure of *Siren*. To sum up those features, one of them is an audio mixer implementation to *Siren* to extend it's connections with SuperCollider, possibly with a *Quark* extension.

Along with interface-level improvements and the visual editor, I am preparing another conference paper for NIME 2018 in collaboration with a media artist, *Mert Toka*. On top of editing the pattern within the *Pattern Roll*(see Sec. 2.1.3.6), it will also be possible to save it as a phrase to be used in the tracker interface (see Sec. 2.8). These sequences will be stored with in a *phrase dictionary* and triggered within the tracker interface by calling the appropriate sequence names. I think that it would be interesting to have a feedback system between patterns where it can also be used as an 'overdubbing' mechanism on top of the algorithmic patterns.

In the other hand, I think that the way forward to develop a competitive hybrid software is through coding it in a native language to be able to fully utilize computer's power. More robust environments such as QT<sup>1</sup> or JUCE<sup>2</sup> would be more suitable for *Siren* as *Javascript* runs on a single thread. It is also possible to develop an integrated pattern language and a sound engine which is tailored to be used with this language. It would also be much more intuitive for non-programmers as the initial setup requirements are reduced. In its current state, a tool like Electron<sup>3</sup> would allow for an easier end-user installation process by packaging the system into a standalone application. In theory, it is also possible to support any number of pattern languages within the *Siren* interface, such as FoxDot<sup>4</sup>.

---

<sup>1</sup>qt.io

<sup>2</sup>juce.com

<sup>3</sup>electronjs.org

<sup>4</sup>foxdot.org

My vision with *Siren* is to create a DAW where the whole context is just a set of scripts on top of some low level code for building out UIs and generating/manipulating audio. Concretely, a more interactive user interface that makes the modules into selectable and manipulable objects could also benefit the experience.



# References

- [1] Ableton. Link. <https://github.com/Ableton/link>, 2016.
- [2] Julian Rohrerhuber Alex McLean. Superdirt. <https://github.com/musikinformatik/SuperDirt>, 2015.
- [3] Algorave. Algorave 5th birthday stream. <https://www.youtube.com/watch?v=g2dINLvLr1g&t=20>. Accessed: 2017-12-19.
- [4] Teresa Amabile. *Componential theory of creativity*. Harvard Business School Boston, MA, 2012.
- [5] HornerImran Amrani. Run the code: is algorave the future of dance music? *The Guardian*, Oct 2017.
- [6] Renick Bell. An approach to live algorithmic composition using conductive. *Proceedings of LAC 2013*, 2013.
- [7] Renick Bell. Experimenting with a generalized rhythmic density function for live coding. In *Linux Audio Conference*, 2014.
- [8] Kim Bjorn. Push turn move. <https://www.pushturnmove.com/>. Accessed: 2017-12-19.
- [9] Alan Blackwell and Nick Collins. The programming language as a musical instrument. *Proceedings of PPIG05 (Psychology of Programming Interest Group)*, 3:284–289, 2005.
- [10] Alan F Blackwell, Thomas RG Green, and Douglas JE Nunn. Cognitive dimensions and musical notation systems. In *Proceedings of International Computer Music Conference, Berlin*, 2000.

- [11] Karen Collins. *Game sound: an introduction to the history, theory, and practice of video game music and sound design*. Mit Press, 2008.
- [12] Karen Collins, Bill Kapralos, Holly Tessler, Chris Nash, and Alan F. Blackwell. Flow of creative interaction with digital music notations.
- [13] Edsger Wybe Dijkstra. Notes on structured programming, 1970.
- [14] Kevin Driscoll and Joshua Diaz. Endless loop: A brief history of chiptunes. *Transformative Works and Cultures*, 2, 2009.
- [15] eightbitbubsy. *ProTracker*. <https://sourceforge.net/projects/protracker/>, 2017. Accessed: 2017-03-30.
- [16] Node.js Foundation. *Node.js*. <http://nodejs.org/>, 2018. Accessed: 2017-03-30.
- [17] francois. A brief history of max. [http://freesoftware.ircam.fr/article.php3?id\\_article=5](http://freesoftware.ircam.fr/article.php3?id_article=5), 2009. Accessed: 2017-11-27.
- [18] Anastasia Georgaki. The grain of xenakistechnological thought in the computer music research of our days. In *Definitive Proceedings of the International Symposium Iannis Xenakis*, 2005.
- [19] Google. Firebase. <https://firebase.google.com/>. Accessed: 2017-12-19.
- [20] T. R. G. Green and M. Petre. Usability analysis of visual programming environments: a ‘cognitive dimensions’ framework. *JOURNAL OF VISUAL LANGUAGES AND COMPUTING*, 7:131–174, 1996.
- [21] Thomas Green and Alan Blackwell. Cognitive dimensions of information artefacts: a tutorial. In *BCS HCI Conference*, volume 98, 1998.
- [22] Andrew Horner and David E Goldberg. Genetic algorithms and computer-assisted music composition. *Urbana*, 51(61801):437–441, 1991.
- [23] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1. ACM, 2007.
- [24] Facebook Inc. *React*. <https://facebook.github.io/react/>, 2018. Accessed: 2017-03-30.

- [25] Can Ince and Mert Toka. Siren: Hierarchical composition interface. In *Proceedings of International Computer Music Conference, Shanghai*, 2017.
- [26] SL Peyton Jones, Cordy Hall, Kevin Hammond, Will Partain, and Philip Wadler. The glasgow haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, volume 93, 1993.
- [27] J. Jong. *Math.js*. <https://github.com/josdejong/mathjs>, 2018. Accessed: 2017-03-20.
- [28] D Knuth. The art of computer programming 1: Fundamental algorithms 2: Seminumerical algorithms 3: Sorting and searching. *MA: Addison-Wesley*, page 30, 1968.
- [29] Charles W Krueger. Software reuse. *ACM Computing Surveys (CSUR)*, 24(2):131–183, 1992.
- [30] Henri Lefebvre. *Rhythmanalysis: Space, time and everyday life*. A&C Black, 2004.
- [31] Justin London. *Hearing in time: Psychological aspects of musical meter*. Oxford University Press, 2012.
- [32] mandarin. *NoiseTracker V2.0 by Mahoney & Kaktus*. <http://www.pouet.net/prod.php?which=13360>, 2018. Accessed: 2017-03-30.
- [33] James McCartney. Supercollider: a new real time synthesis language. In *Proc. International Computer Music Conference (ICMC96)*, 1996.
- [34] James McCartney. Rethinking the computer music language: Supercollider. *Computer Music Journal*, 26(4):61–68, 2002.
- [35] Alex McLean. Toplap. <https://toplap.org/>. Accessed: 2017-12-19.
- [36] Alex McLean. The textural x. *Proceedings of xCoAx2013: Computation Communication Aesthetics and X*, pages 81–88, 2013.
- [37] Alex McLean. Making programming languages to dance to: live coding with tidal. In *Proceedings of the 2nd ACM SIGPLAN international workshop on Functional art, music, modeling & design*, pages 63–70. ACM, 2014.

- [38] Alex Mclean and Geraint Wiggins. Bricolage programming in the creative arts. *22nd Psychology of Programming Interest Group*, 12 2010.
- [39] Alex McLean and Geraint Wiggins. Tidal–pattern language for the live coding of music. In *Proceedings of the 7th sound and music computing conference*, 2010.
- [40] Alex McLean and Geraint A Wiggins. Texture: Visual notation for live coding of pattern. In *ICMC*, 2011.
- [41] Christopher Alex McLean et al. *Artist-programmers and programming languages for the arts*. PhD thesis, Goldsmiths, University of London, 2011.
- [42] John S Mill. A system of logic ratiocinative and inductive london. *Robson, JM (Hg.), Collected Works of John Stuart Mill*, 7, 1843.
- [43] George A. Miller. Wordnet: A lexical database for english. *Commun. ACM*, 38(11):39–41, November 1995.
- [44] Chris Nash. The cognitive dimensions of music notations. In *The Cognitive Dimensions of Music Notations*, 2015.
- [45] Chris Nash and Alan Blackwell. Tracking virtuosity and flow in computer music. In *ICMC*, 2011.
- [46] Chris Nash and Alan Blackwell. Liveness and flow in notation use. In *NIME*, 2012.
- [47] George Papadopoulos and Geraint Wiggins. Ai methods for algorithmic composition: A survey, a critical view and future prospects. In *AISB Symposium on Musical Creativity*, pages 110–117. Edinburgh, UK, 1999.
- [48] Jeff Pressing. Black atlantic rhythm: Its computational and transcultural foundations. *Music Perception: An Interdisciplinary Journal*, 19(3):285–310, 2002.
- [49] Miller Puckette et al. Pure data: another integrated computer music environment. *Proceedings of the second intercollege computer music concerts*, pages 37–41, 1996.
- [50] Miller Puckette, David Zicarelli, et al. Max/msp. *Cycling*, 74:1990–2006, 1990.
- [51] Claude Rostand. Metastasis. *Iannis Xenakis: Metastasis, Pithoprakta, Eonta, recording (New York, NY: Vanguard Recoding Society)*, 1967.

- [52] Chris Sattinger. Supercolliderjs. <https://crucialfelix.github.io/supercolliderjs/>.
- [53] Mary Simoni. Algorithmic composition: a gentle introduction to music composition using common lisp and common music. *SPO Scholarly Monograph Series*, 2003.
- [54] C.J. Sippl and C.P. Sippl. *Computer dictionary and handbook*. H. W. Sams, 1972.
- [55] L Spiegel. Distinguishing random, algorithmic, and intelligent music. *Internet: http://retiary.org/ls/writings/alg-comp-ltr-to-cem.html*, 1989.
- [56] Paul Tingen. Autechre. <https://www.soundonsound.com/people/autechre>, 2004.
- [57] Raymond Turner and Amnon H Eden. The philosophy of computer science: introduction to the special issue, 2007.
- [58] Peter Worth and Susan Stepney. Growing music: musical interpretations of l-systems. In *Workshops on Applications of Evolutionary Computation*, pages 545–550. Springer, 2005.
- [59] Matthew Wright. Open sound control: an enabling technology for musical networking. *Organised Sound*, 10(3):193–200, 2005.
- [60] Matthew Wright, Adrian Freed, et al. Open soundcontrol: A new protocol for communicating with sound synthesizers. In *ICMC*, 1997.
- [61] I. Xenakis. *Formalized Music: Thought and Mathematics in Composition*. Harmonologia series. Pendragon Press, 1992.