CODE SCORES IN LIVE CODING PRACTICE

Thor Magnusson
Department of Music
University of Sussex
Brighton, BN1 9RH, UK
t.magnusson@sussex.ac.uk

ABSTRACT

This paper explores live coding environments in the context of notational systems. The improvisational practice of live coding as combining both composition and performance is introduced and selected systems are discussed. The author's Threnoscope system is described, but this is a system that enables the performer to work with both descriptive and prescriptive scores that can be run and altered in an improvisational performance.

1. INTRODUCTION

The live coder sits on stage and writes software in front of a live audience. The desktop is projected on the wall in a gesture of sharing and audience engagement [1, 2]. In the past decade, live coding has become a popular performance practice, supported by the diverse interpreted and high level programming languages that suit the practice. Furthermore, the popular hacker and maker cultures are affecting general culture such that coding is now considered a creative activity on par with drawing or playing an instrument. The live coding community has played an important role here and been active in disseminating the practice by sharing code, organizing festivals and conferences, and establishing research networks.

Code is a form of notation that works extremely well in musical composition, especially when the aim is to write non-linear, interactive, or context aware music [3]. Although general-purpose languages can be used for musical live coding, many live coders have created their own mini-languages for a particular performance style, genre, or even a performance. The new language becomes an instrument, a framework for thinking, with strong considerations of notational design. Here, language designers have invented graphical interfaces like we find in Pure Data or Max/MSP; game interfaces, as in Dave Griffiths' *Al Jazaari*; functional notation, like McLean's *Tidal*; or Chris Kiefer's physical controllers that encode genetic algorithms of sound synthesis [4].

2. NOTATION AND INTERPRETATION

Notation is a way of communicating abstract ideas to an interpreter, and in live coding that interpreter is typically a compiler called the "language interpreter." Standard Music Notation is a system of notation that has developed from the general recognition that symbols can capture more information, coherent in meaning between composers, interpreters and cultures, in a smaller space than natural language or bespoke new symbolic languages. Standard Music Notation is a cognitive tool that has developed with requirements for a standard language and concerns about sight-reading and rapid understanding. Composers are able to rely on the performer's expertise and creative interpretation skills when the piece is played. Conversely, in the symbolic notation for computer music composition and performance, we encounter an important difference in the human and the computer capacity for interpretation: the human can tolerate mistakes and ambiguity in the notation, whereas the computer cannot. Natural language programming of computers is clearly possible, for example:

```
produce a sine wave in A
name this synth "foo"
wrap this in an ADSR envelope
repeat foo four times per second
name this pattern "ping"
```

However, the problem here is one of syntax: what if the coder writes "Sine" instead of "sine," "440" instead of "A," or "every 0.25 second" instead of "four times per second?" The cognitive load of having to write natural language with the programming language's unforgiving requirements for perfect syntax makes the natural language approach less appealing than writing in traditional programming languages, for example in functional or object orientated languages. Of course, semi-natural language programming languages have been invented, such as COBOL, Apple Script, or Lingo. The problems with those were often that they became quite verbose and the 'naturalness' of their syntax was never so clear. Consequently, in a more familiar object oriented dot-syntax, the above might look like:

```
w = Sine("foo", [\freq, 440]);
w.addEnvelope(\adsr);
p = Pattern("ping");
p.seq(\foo, 0.25);
```

In both cases we have created a synthesizer and a pattern generator that plays the synth. The latter notation is less prone to mistakes, and for the trained eye the syntax actually becomes symbolic through the use of dots, camelCase, equals signs, syntax coloring, and brackets with arguments that are differently formatted according to type. This is called 'secondary notation' in computer science parlance, and addresses the cognitive scaffolding offered by techniques like colorization or indentation [5].

3. LIVE CODING AS NOTATION

Live coding is a real-time performance act and therefore requires languages that are relatively simple, forgiving in terms of syntax, and high level. Certain systems allow for both high and low level approach to musical making, for example SuperCollider, which enables the live coder to design instruments (or synths) whilst playing them at another level of instructions (using patterns). Perhaps the live coding language with the most vertical approach would be Extempore [6], which is a live coding environment where the programming language Scheme is used at the high level to perform and compose music, but another language – type sensitive and low level, yet keeping the functional programming principles of Scheme – can be used for low level, real-time compiled instructions (using the LLVM compiler). In Extempore, an oscillator, whether in use or not, can be redesigned and compiled into bytecode in real-time, hotswapping the code in place.

However, live performance is stressful and most live coders come up with their own systems for high-level control. The goals are typically fast composition cycle, understandability, novel interaction, but most importantly to design a system that suits the live coder's way of thinking. Below is an introduction of four systems that all explore a particular way of musical thinking, language design, and novel visual representation.

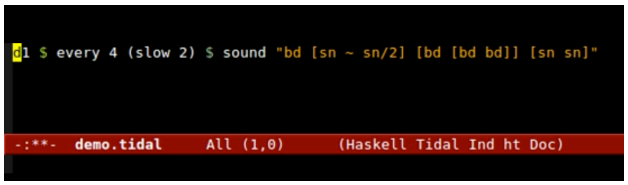

Figure 1. A screen shot of Tidal. We see the score written in the quotation marks, with functions applied.

Alex McLean's *Tidal* [7] is a high level mini-language built on top of Haskell. It offers the user a limited set of functionality; a system of constraints that presents a large space for exploration within the constraints presented [8]. The system focuses on representing musical pattern. The string score is of variable length, where items are events, but these items can be in the form of multi-dimensional arrays, representing sub-patterns. This particular design decision offers a fruitful logic of polyrhythmic and polymetric temporal exploration. The system explicitly *affords* this type of musical thinking, which consequently limits other types of musical expression. The designers of the live coding languages discussed in this section are not trying to create a universal solution to musical expression, but rather define limited sets of methods that explore certain musical themes and constraints.

Dave Griffiths' *Scheme Bricks* is a graphical coding system of a functional paradigm, and, like Tidal, it offers a way of creating recursive patterns. Inspired by the MIT Scratch [9] programming system, a graphical visualization is built on top of the functional Scheme programming language. The user can move blocks around and redefine programs through visual and textual interactions that are clear to the audience. The colored code blocks are highlighted when the particular location of the code runs, giving an additional representational aspect to the code.

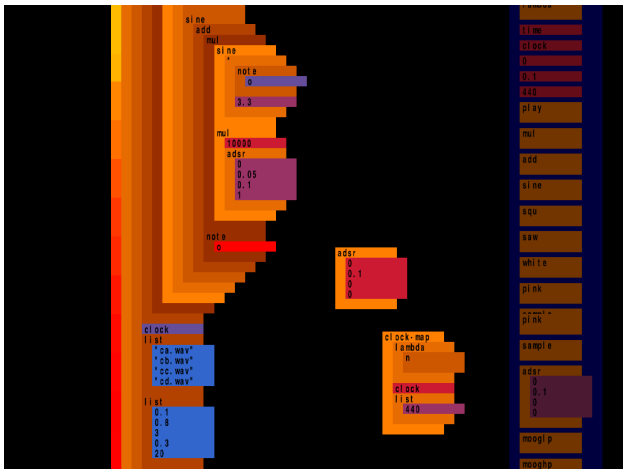

Figure 2. Scheme Bricks. In simple terms, what we see is the bracket syntax of Scheme represented as blocks.

Scheme Bricks are fruitful for live musical performance as patterns can be quickly built up, rearranged, muted, paused, etc. The modularity of the system makes it suitable for performances where themes are reintroduced (a muted block can be plugged into the running graph).

This author created *ixi lang* in order to explore code as musical notation [10]. The system is a high level language built on top of SuperCollider and has access to all the functionality its host. By presenting a coherent set of bespoke 'ixi lang' instructions in the form of a notational interface, the system can be used by novices and experienced SuperCollider users alike. The system removes many of SuperCollider's complex requirements for correct syntax, whilst using its synth definitions and patterns; the original contribution of ixi lang is that it creates a mini-language for quick prototyping and thinking.

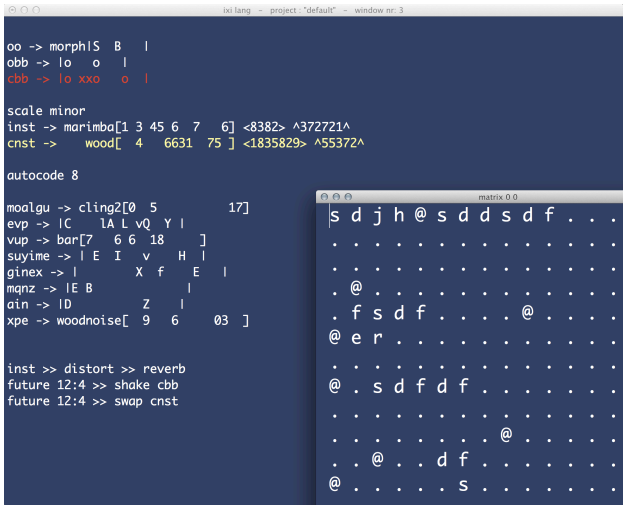

Figure 3. ixi lang Agents are given scores that can be manipulated, and changed from other code.

In ixi lang the user creates agents that are assigned percussive, melodic, or concrete scores. The agents can be controlled from other locations in the code and during that process the textual document is automatically rewritten to reflect what is happening to the agents. This makes it possible for the coder and the audience to follow how the code is changing itself and the resulting music. As code can be rewritten by the system, it also offers the

possibility of storing the state of the code at any given time in the performance. This is done by writing a snapshot with a name: the snapshot can then be recalled at any time, where running new code is subsequently muted (and changes color), and agents whose score has changed return to their state when the snapshot was taken.

A recent live coding environment by Charlie Roberts called *Gibber* [11] takes this secondary notation and visual representation of music further than ixi lang: here we can see elements in the code highlighted when they are played: the text flashes, colors change, and font sizes can be changed according to what is happening in the music. Gibber allows for any textual element to be mapped to any element in the music. The code becomes a visualization of its own functionality: equally a prescription and description of the musical processes.

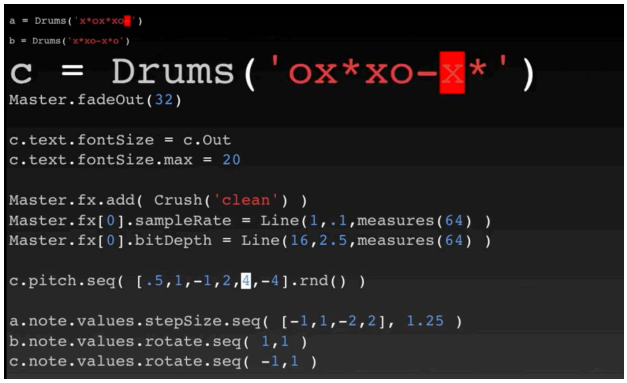

Figure 4. Gibber. Here textual code can change size, color or font responding to the music. All user-defined.

Gibber is created in the recent Web Audio API, which is a JavaScript system for browser-based musical composition. As such it offers diverse ways of sharing code, collaborating over networks in real-time or not.

All of the systems above use visual elements as both primary and secondary notation for musical control. The notation is prescriptive – aimed at instructing computers – although elements of secondary notation can represent information that could be said to be of a descriptive purpose [12]. The four systems have in common the constrained set of functions aimed to explore particular musical ideas. None of them – bar Gibber perhaps – are aimed at being general audio programming systems, as the goals are concerned with live coding: real-time composition, quick expression, and audience understanding.

4. THE THRENOSCOPE

In the recent development of the *Threnoscope* system, the author has explored representational notation of live coding. This pertains to the visualization of sound where audible musical parameters are represented graphically. The system is designed to explore microtonality, tunings, and scales; and in particular how those can be represented in visual scores aimed at projection for the audience.

The Threnoscope departs from linear, pattern-based thinking in music and tries to engender the conditions of musical stasis through a representation of sound in a circular interface where space (both physical space and pitch space) is emphasized, possibly becoming more important than concerns of time.

The system is object oriented where the sound object – the 'drone' – gets a graphical representation of its state. This continuous sound can be interacted with through code, the graphical user interface, MIDI controllers, and OSC commands, and changes visually depending upon which parameters are being controlled. The user can also create 'machines' that improvise over a period of time on specific sets of notes, as defined by the performer. These machines can be live coded, such that their behavior changes during their execution. Unlike the code score, discussed below, the machines are algorithmic: they are not intended to be fully defined, but rather to serve as an unpredictable accompaniment to the live coder.

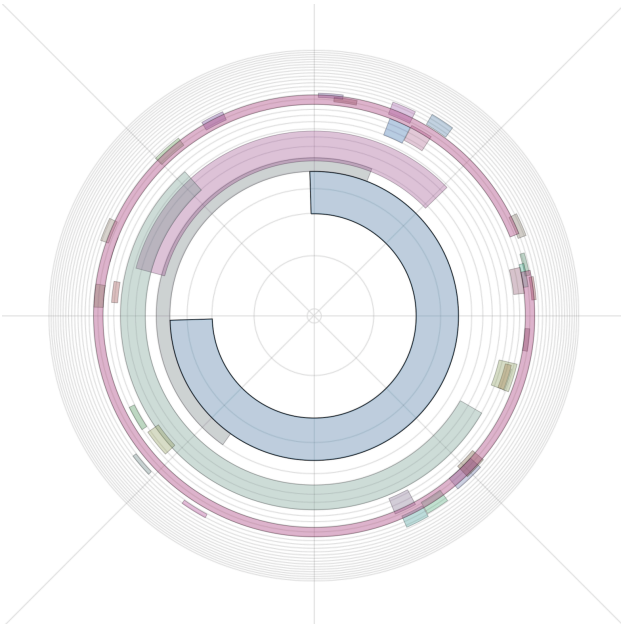

Figure 5. The Threnoscope in 8-channel resolution. The straight crossing lines are speakers. The circles are harmonics, and the colored wedges are (moving) drones.

Figure 5 shows the circular interface where the innermost circle is the fundamental frequency, with the harmonics repeated outwardly. The lines crossing the interface represent the audio channels or speakers (the system can be set from 2 to 8 channels). The sound/drone can have a length extending up to 360 degrees, but it can also be short and move fast around the space. Figure 6 depicts the system with the command line prompt on the right, and a console underneath that reports on the state of the engine, errors in code, or events being played in a running score. By clicking on a particular drone, its sonic information appears in the console in a format that gives the coder quick entry to manipulate the parameters.

Musical events in the Threnoscope system are created through code instructions. Since the default envelope of the drone is an ASR (Attack, Sustain, Release) envelope, a note duration can range from a few milliseconds to an infinite length. Each of the speaker lines could be seen as a static playhead, where notes either cross it during their movement or linger above it with continuous sound. A

compositional aspect of the Threnoscope is to treat notes as continuous objects with states that can be changed (spatial location, pitch, timbre, amplitude, envelope, etc.) during its lifetime.

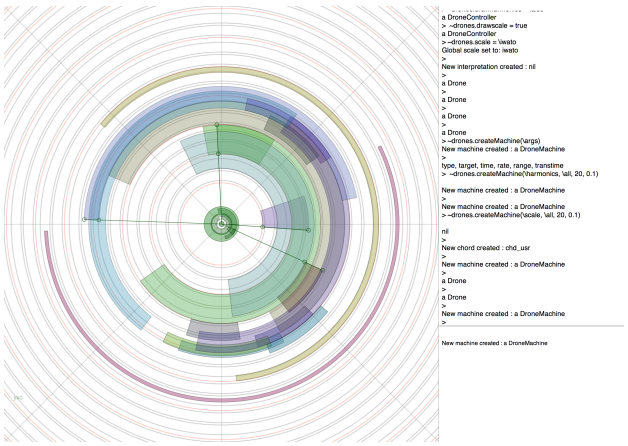

Figure 6. The Threnoscope's code interface on the right. The system is here in a scale-mode, with scale degrees rather than harmonics. A machine is running in the middle, affecting a selection of the running drones.

The Threnoscope has been described before both in terms of musical notation [11] and as a system for improvisation [12]. This paper explores further the notational aspects of the system, and the design of the code score.

5. THE CODE SCORE

Code is rarely represented on a timeline, although certain systems have enabled programmers to organize code linearly over time, although in Macromedia's Director and Flash multimedia production software this becomes a key feature. This general lack of a timeline can pose a problem when working with code as a creative material in time-based media such as music, games or film. The lack of timeline makes navigating the piece for compositional purposes cumbersome and often impossible.

The Threnoscope's code score is a two dimensional textual array where the first item is the scheduled time and the second contains the code to be executed. This makes it possible to jump to any temporal location in the piece, either directly or through running the code that is scheduled to happen before (with some limitations though, as this code could be of a temporal nature as well).

Scores in textual code format, like that of the Threnoscope, can make it difficult to gain an overview of the musical form, as multiple events can be scheduled to take place at the same moment with subsequent lack of activity for long periods. This skews the isomorphism between notational space (the lines of code) and time if working with the mindset of a linear timeline. For this reason the Threnoscope offers an alternative chronographic visualization to represent the code in the spatial dimension. This is demonstrated in Figure 7.

The code score timeline is vertically laid out as is common in tracker interfaces. The code can be moved around in time, deleted, or new elements added. By clicking on relevant 'code tracks' the user can call up code into a text field and edit the code there. The drones are created on the vertical tracks on the timeline. They have a beginning and an end, with code affecting the drones represented as square blocks on the drone track. The drone itself and the events within it can be moved around with the mouse or through code. The score can therefore be manipulated in real-time, much like we are used to with MIDI sequencers or digital audio workstations.

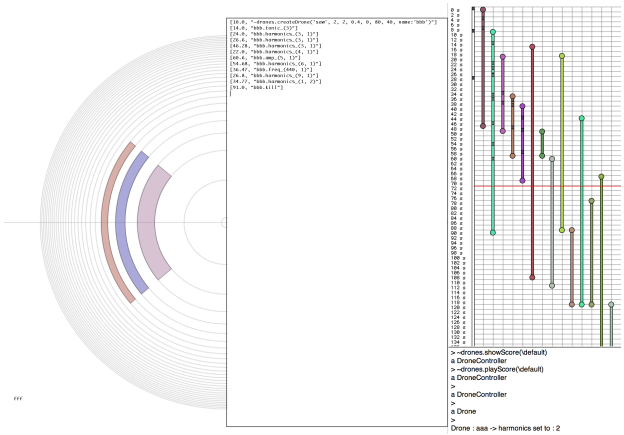

Figure 7. A graphical visualization of the code score. When a vertically lined drone is clicked on, a page with its code appears above the circular interface.

Most timelines in music software run horizontally from left to right, but in the Threnoscope the score vertical and runs from top down. This is for various reasons: firstly, the available screen space left on most display resolutions when the circular score has taken up the main space on the left is a rectangular shape with the length on the vertical axis; secondly, when a user clicks on the visual representation of the drone, its score pops up in the textual form of code, and this text runs from top to bottom. It would be difficult to design a system where code relates to events on a horizontal timeline.

6. PERFORMING WITH SCORES

The code score was implemented for two purposes: to enable small designed temporal patterns to be started at any point in a performance: just like jazz improvisers often memorize certain musical phrases or licks, the code score would enable the live coder to pre-compose musical phrases. The second reason for designing the code score system is to provide a format for composers to write longer pieces for the system, both linear and generative.

The score duration can therefore range from being a short single event to hours of activity; it can be started and stopped at any point in a performance, and the performer can improvise on top of it. Scores can include other scores. As an example, a performer in the middle of a performance might choose to run a 3-second score that builds up a certain tonal structure. The code below shows the code required to start a score.

```
~drones.playScore(\myScore, 1) // name of score & time scale
~drones.showScore(\myScore) // visual display of the score
```

The first method simply plays the score without a graphical representation. This is very flexible, as multiple scores can be played simultaneously, or the same score started at different points in time. Scores can be stopped at will. Whilst the scores are typically played without any visual representation, it can be useful to observe the score graphically. The second method creates the above-mentioned graphical representation of the score shown in Figure 7. For a live performance, this can be helpful as it allows the performer to interact with the score during execution. The visual representation of the score can also assist in gaining an overview of a complex piece.

For this author, the code score has been a fruitful and interesting feature of the system. Using scores for digital systems aimed at improvisation becomes equivalent to how instrumentalists incorporate patterns into their motor memory. The use of code scores question the much broken unwritten 'rule' that a live coding performance should be coded from scratch. It enables the live coder work at a higher level, to listen more attentively to the music (which, in this author's experience, can be difficult when writing a complex algorithm), and generally focus more on the compositional aspects of the performance.

7. CONCLUSION

This short paper has discussed domain specific programming languages as notational systems. Live coding systems are defined as often being idiosyncratic and bespoke to their authors' thought processes. The Threnoscope and its code score was presented as a solution to certain problems of performance and composition in live coding, namely of delegating activities to actors such as machines or code scores.

8. REFERENCES

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

[2] T. Magnusson. "Herding Cats: Observing Live Coding in the Wild*"* in *Computer Music Journal*, vol. 38 no. 1. pp. 8-16. 2014.

[3] T. Magnusson. "Algorithms as Scores: Coding Live Music" in *Leonardo Music Journal*. vol 21. no. 1. pp. 19-23. 2011.

[4] C. Kiefer. "Interacting with text and music: exploring tangible augmentations to the live-coding interface" in *Proceedings of the International Conference for Life Interfaces*. 2014.

[5] A. F. Blackwell, and T. R. G. Green. "Notational systems - the Cognitive Dimensions of Notations framework" in J.M. Carroll (Ed.) *HCI Models, Theories and Frameworks: Toward a multidisciplinary science*. San Francisco: Morgan Kaufmann, pp. 103-134. 2003.

[6] A. Sorensen, B. Swift, and A. Riddell. "The Many Meanings of Live Coding" in *Computer Music Journal*. vol. 38. no. 1. pp. 65-76. 2014.

[7] A. McLean. "Making programming languages to dance to: Live coding with Tidal" in *Proceedings of the 2nd ACM SIGPLAN International Workshop on Functional Art, Music, Modelling and Design*. 2014.

[8] T. Magnusson. "Designing constraints: composing and performing with digital musical systems" in *Computer Music Journal*, vol. 34. No. 4. pp. 62-73. 2010.

[9] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. S. Silverman, and Y. Kafai. "Scratch: Programming for All" in *Communications of the ACM*, vol. 52. no, 11. 2009.

[10] T. Magnusson. "ixi lang: a SuperCollider parasite for live coding" in *International Computer Music Conference*, 2011.

[11] C. Roberts, and J. Kuchera-Morin. "Gibber: Live Coding Audio in the Browser." in *Proceedings of the International Computer Music Conference*. pp. 64-69. 2012.

[12] A.F. Blackwell, and T.R.G. Green. "Notational systems - the Cognitive Dimensions of Notations framework" in J.M. Carroll (Ed.) *HCI Models, Theories and Frameworks: Toward a multidisciplinary science*. San Francisco: Morgan Kaufmann, pp. 103-134. 2003.

[13] T. Magnusson. "Scoring with code: composing with algorithmic notation" in *Organised Sound*, vol. 19. no. 3. pp. 268-275. 2014.

[14] T. Magnusson. "Improvising with the threnoscope: integrating code, hardware, GUI, network, and graphic scores" in *Proceedings of the New Interfaces for Musical Expression Conference*. 2014.