# CONFESSIONS OF A LIVE CODER

*Thor Magnusson*
ixi audio &
Faculty of Arts and Media
University of Brighton
Grand Parade, BN2 0JY, UK

**ABSTRACT**

This paper describes the process involved when a live coder decides to learn a new musical programming language of another paradigm. The paper introduces the problems of running comparative experiments, or user studies, within the field of live coding. It suggests that an autoethnographic account of the process can be helpful for understanding the technological conditioning of contemporary musical tools. The author is conducting a larger research project on this theme: the part presented in this paper describes the adoption of a new musical programming environment, Impromptu [35], and how this affects the author's musical practice.

## 1. INTRODUCTION

A prominent discourse exists within the philosophy of technology regarding how the tools we use define our activities [14, 38]. The neutrality of technology has been strongly questioned [18]. Although rarely discussed or analysed, practitioners and researchers in the field of computer music are conscious of how specific musical environments encourage certain practices and prevent others. Connoisseurs report that they can identify certain musical environments, not only by how they sound, but also by which musical patterning or form they afford. Each musical programming language or software has its own functional character and the user learns to think according to its ways [24]. In the research context of how this technological conditioning takes place, live coding presents itself as an ideal field for investigation due to the strong dependency on technology and the available programming languages in which to think.

But how would one *study* this influence of the tool or programming language upon the musician? Experiments could be set up where experimental groups and control groups perform some musical tasks in the diverse programming environments, and then analyse how the results vary. Unfortunately, this proves to be difficult for two key reasons: a) the programming environments are highly complex and based on diverse paradigms of thinking. It would be hard to find test subjects with similar programming backgrounds. b) computer music is an intricate and convoluted field where the test subject's knowledge ranges from sound synthesis, acoustics, psychoacoustics, musical theory, to computational creativity. Finding people with comparable backgrounds is a near impossible task. Additionally, musical goals differ profoundly between any two practitioners: people typically choose to embark upon writing music with programming languages for very specific reasons and those are rarely comparable.

At ICMC 2007, in Copenhagen, I met Andrew Sorensen, the author of Impromptu and member of the aa-cell ensemble that performed at the conference. We discussed how one would explore and analyse the *process of learning* a new programming environment for music. One of the prominent questions here is how a functional programming language like Impromptu would influence the thinking of a computer musician with background in an object orientated programming language, such as SuperCollider? Being an avid user of SuperCollider, I was intrigued by the perplexing code structure and work patterns demonstrated in the aa-cell performances using Impromptu [36]. I subsequently decided to embark upon studying this environment and perform a reflexive study of the process.

This paper is a report of a larger research project on technological conditioning, in this instance with a particular focus on how learning a new music programming environment influences ideas in the areas of composition and performance. The investigation uses qualitative research methods to achieve this, through two strands of enquiry: a) an autoethnographic and phenomenological first-person account of the author's own experience, and b) the study and interviewing of student and workshop participants learning the same programming environment. Self-observation and analysis as described in a) is not common in the field of music, but there are precedents [3, 39].

## 2. THE AUTOETHNOGRAPHIC METHOD

Whilst scholarly first person accounts of experience have existed for millenia – a good example being the Confessions of St. Augustine, finished in AD 398 and discussed in the highly reflexive philosophy of Ludwig Wittgenstein in the 1940s [43] – it has been frowned upon within the field of science, and for good reasons, although scientific objectivity has been strongly questioned by philosophers of science [10, 16]. There are situations, and indeed discourses, where reflective first person accounts can give deeper insights, clearer analysis, and better interpretation than achievable with traditional objective scientific methodologies. This is acknowledged in the diverse disciplines [7, 10, 31, 41].

Autoethnography seeks to enable such subjective investigation through a formal research methodology. It has been widely used in fields ranging from medicine, for example where the ethnographer writes about death and dying [17]; to issues of race, where a black Jew describes his complicated identity [2]; to anthropology

where, as opposed to traditional ethnography, the ethnographer actually reports on the experience of being part of a new culture [1]. It originates from ethnography and the realisation within that field that the ethnographer's persona and interpretation is always present in the data collection, analysis, and interpretation of the object of study. One reaction to this realisation is to acknowledge the personal presence and actually discuss, in the first person, the researcher's experience during the data collection and interpretation. Although giving space for the subjective, autoethnography thus adheres to the systematic methodologies developed in qualitative research, enabling the output to be more than a mere autobiography.

I chose to use autoethnography due to the stated problems of running reasonably controlled experiments in studying the effects of different programming languages on the musical output of such a diverse and specialised group of people as live coders are. Polanyi's work [31] on how knowledge is always personal, involving tacit dimensions that are unique to the individual, is pertinent to this project's research statement. A method like autoethnography enables the researcher to make some of that explicit. Some authors argue that such a first person approach could be of general benefit as a methodological tool in computer science [12]. However, the current study is broader than being a solely first person account, as I teach creative music programming at a university level, give workshops internationally, and am part of a wider community of artists that work with programming languages as their primary material. The collected data derives from conversations, surveys [20, 21], and teaching.

I started my study of Impromptu in June 2010 and have therefore been working with it for nearly a year, as much as a busy professional life allows. I kept a journal of the learning process, communicated on the Impromptu mailing list, and set up a research blog [23]. I have introduced Impromptu in sessions of computer music and live coding, and have taken notes of how new learners deal with this new language that is often strongly foreign, since it is rare encountering people that have studied functional programming.

## 3. SCORING ALGORITHMS

Live coding needs no introduction [4, 5, 6, 29]. It has become so prominent as a practice within computer music, that ICMC 2011 has included it explicitly as a submission category. In live coding performances we witness the results of the performer's habituation with a chosen programming language that enables the coder to think in specific ways. It is an incorporation of programming styles, thought patterns, and solutions. Just as the guitarist embodies the "riff," the live coder has assimilated the algorithm. Knowledge of the language syntax and semantics, as well as practiced problem solving, and the array of learned algorithms becomes the toolset or the framework in which the live coder can think.

Establishing new frameworks of thinking by learning new programming environments is a familiar experience to most programmers. What is novel in live coding is the real-time nature of writing code and the improvisation of computer music. Additionally, projecting programming code onto the wall of a club with the audience following the evolution of the composition is also innovative. Programming becomes a performance art with music and/or visual art as its subject. Moreover, the performers typically use idiosyncratic systems that are often works of art themselves [11]. New criteria emerge in the design of programming languages, e.g., regarding how lay people can engage with the code.

Live coding is exciting. It forces the composer to reveal his/her compositional thought pattern, to make public an intimate process that might result in profound successes or dire mistakes. Moreover, as the coding is typically an improvisation, it means that the performer is rarely able to foresee the result of the performance. The act of programming obviously requires strong attention to details; a misplaced comma or a bracket will result in a failure of execution. Even if highly rewarding, live coding is therefore a stressful activity that requires strong focus, practice and luck!

My own live coding practice started with using SuperCollider as part of improvisation groups with acoustic and electric instruments. Gradually I became interested in creating a higher level abstraction for such improvised playing which resulted in the ixi lang [22]. The idea was to abstract away as much procedural thinking as possible, moving towards a more declarative style of programming. The ixi lang has proved successful in many ways, but I increasingly wanted more flexibility and became interested in being challenged with new work patterns. I therefore decided to explore other systems.

## 4. THINKING THROUGH TOOLS

Everyone that has tried programming a computer knows that there is no magic involved: the language consists of strict semantic elements and syntactic rules. There are atoms, lists, brackets and curly brackets, full stops, commas, semi-colons and colons, and keywords that perform specific functions. Although performance speed is a primary consideration, most languages aim at scarcity and readability, focusing on how the language design can afford powerful ways of thinking in terms of organisation and manipulation of data. Languages differ in paradigm: the flow of data through functions in Haskell might feel natural to one coder, whilst the object orientated approach presented by Java might be more appropriate to another's way of thinking.[1] Also of consideration is layout and style: whilst Scheme might confuse newcomers with an abundance of brackets, Python might frustrate others by relying on tabs and spaces to inline code (i.e., layout becomes syntactic).

[1] Conference series and interest groups are dedicated on the psychology of programming, e.g., www.ppig.org.

In most cases, spatial organization or colouring of code is a secondary syntax/notation [27]. It does not affect how the computer understands the instructions, but it has various functions for the creator and other human readers of the code, the audience in the case of live coding. In a context where the audience might be unfamiliar with the system used, it is important to be able to indicate what the key elements of the system are, such as key musical functions, synthesizers, pattern generators; classes, variables, arguments or comments. This can be done through text inlining, syntax colourisation, capitalisation or font size, and special symbols instructing the interpreter to ignore text written as comments. In live coding the visual component is an important element in connecting with the audience. As McLean et al. show there are systems where the visual become *primary* syntax for code interpreters. Most dataflow languages, such as Pure Data, do not fall into this category, as visual layout does not affect function. However, systems like Scheme Bricks [27], Texture [26], Scratch [27] and more do indeed rely on the spatial to organise code. (See Figure 1).

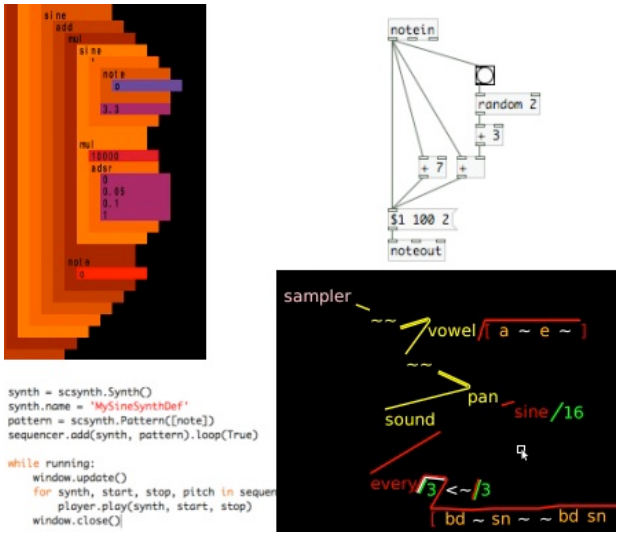


**Figure 1**. A collage of four different programming languages used in live coding that all address space differently. Clockwise depicting Scheme Bricks, Pure Data, Texture, and Python.

The person embarking upon the practice of live coding music is therefore faced with countless decisions. What language to use, at what level of abstraction, whether to work in the textual or the graphical domain, which sound engine or what musical pattern algorithms? Artists choose to work in languages that can deliver what they intend, but the inverse is also true, that during the process of learning a programming environment, it begins to condition how the artist thinks. As Latour states, there is no shooting without a person pulling the trigger, but neither is there shooting without the gun. What emerges is a hybrid, a *gun-person*, which becomes the actor [18]. Underpinning this research project is my interest in how, by extending my thought patterns through the use of a new tool, I would become a new kind of hybrid, making fresh creative decisions.

## 5. THE WAYS OF THE LANGUAGE

Learning a new programming language for musical creation involves personal adaptation on various fronts: the environment has an unfamiliar culture around it, the language has unique characteristics, and the assimilation process might change ones musical goals. Below I will discuss these three topics.

### 5.1. The Culture of Impromptu

For the learner of any new musical programming language, whether an experienced programmer or not, the availability of tutorials, help files, mailing lists and forums is extremely important. This initial encounter with the tool will either create the perception that there is a straightforward path to progressing and mastering the language, or result in confusion, disappointment, and eventually the resignation to failure.

Impromptu performs this initial invitation very well through its website (http://impromptu.moso.com.au). Since it uses the Scheme programming language, links are provided to free online books on Scheme. There is a clear introduction tutorial that enables the user to create a simple melody within minutes, and other tutorials are divided into topics such as audio, graphics and video, OpenGL coding, and on extending the language with Objective-C. Examples of code are provided both online and distributed with the application, and these will take the user through an engaging journey displaying the power of Impromptu. Most impressively, Sorensen has recorded coding sessions as video screen casts, both step-by-step tutorials and live coding, and placed them online. All together, this makes the experience of learning Impromptu quite pleasurable.

There is no online forum, but a mailing list exists. The list is low traffic and the level of discussion on it shows that the majority of people subscribed are experienced Scheme programmers that have been enticed by Impromptu for its use of that language. There are very few, if any, programming novices active on the list, so questions are posted mostly on the environment and not the language. Compared to the SuperCollider list, the main difference is in the volume of traffic and the speed in which answer to a posted question is received. (Normally minutes on the SuperCollider list, but can take a few days on the Impromptu list). Both lists are very friendly to newcomers.

The high skill entry level of Impromptu might be explained if we look at the history of the relevant programming languages and the culture of its usage. Scheme is a Lisp variant, a functional language that is considered very elegant by many scholars in computer science, but it is rarely used in industry or by self-taught web-based programmers [42]. Many young people today have dabbled with programming through creating their own website, coded in JavaScript, experimented with Flash or Processing, or perhaps bought an "introduction to programming" book, but Scheme is hardly the next logical step on the path to mastery of coding. Functional programming is simply not used

much in the industry, even though it is highly valued by many skilled programmers.

```
;; creating an oscillator
(definec make-oscil
   (lambda (phase)
      (lambda (amp freq)
         (let ((inc (* 3.141592 (* 2.0 (/ freq 44100.0)))))
            (set! phase (+ phase inc))
            (* amp (sin phase))))))

(definec sine-inst
   (sine-inst closure* (closure* double double double double double*))
   (lambda ()
      (let* ((losc (make-oscil 0.0))
             (rosc (make-oscil 0.0)))
         (lambda (freq time channel data)
            (if (> channel 0.0)
                (rosc 0.5 freq)
                (losc 0.5 freq))))))

;; now play the instrument
(define loop
   (lambda (beat)
      (dotimes (i 16)
         (play (* i 1/4) code1 (pc:random 70 90 '(0 2 3 5 7 8)) 80 1/8))
      (for-each (lambda (p)
                  (play (* 3/2 (random 4)) code1 (+ p 12) 80 .5)
                  (play code1 p 80 1))
                (pc:make-chord 40 70 4 (random '((0 3 7) (2 5 8) (7 11 2)))))
      (callback (*metro* (+ beat (* 1/2 4))) 'loop (+ beat 4))))

(loop (*metro* 'get-beat 4))
```

**Figure 2**. Code from the recent introduction to Impromptu 2.5, where signal processing code can be written in real-time.

One of the initial questions I asked myself was how the adaption of a new musical environment would affect my musical habits. In this context, the culture around the environment is important. One goes through tutorials, help files, runs code from the list, experiments, and asks questions on the list. All these activities involve the habituation of the language as applied by experienced users. Learning the environment thus involves the initiation into a specific cultural practice. I found this experience oddly analogous to Sudnow's description of him learning the jazz piano [39], except in my case the learning had less to do with embodiment, and more about learning new ways of thinking.[2] During this process my musical thoughts changed significantly and I found myself trying to achieve new musical goals. Inadvertently I paid less attention to synthesis and sonic texture, but focused more on musical phrases and form. I do think that Impromptu encouraged this change in my practice and this influence comes partly from the culture around the tool, but it is equally clear that the language constructs encourage this as well.

What is immediately noticeable is the lack of user contribution in terms of sharing, posting of code or pieces, or libraries. Scheme is in a way a meta-language that enables easy creation of impressive additions or alternative language structures, so there could be a strong motivation to share one's creations. Many reasons could be the cause of this: there might not be many users who have written such libraries, they might be proud programmers that don't want to publicise imperfect code, or they might be unsure about the quality of their music. Or simply that such sharing hasn't been encouraged on the list (it is notoriously difficult to build up a good mailing list ambiance).

Not to be overlooked is the fact that Impromptu is closed source, which projects one immediately as a user, the receiver of goods, and not a potential developer. There are striking similarities between SuperCollider 2 and Impromptu here. Before 2002, SuperCollider was closed source and had a much smaller user community, where the mailing list was primarily maintained (answering questions and so on) by James McCartney.

Interestingly, Sorensen has recently announced that he is working on a new system, Extempore, which is open source and cross-platform [34]. This is welcomed since it will enable people to become more involved and invest time and learning into a system that will continue developing, gain developer base, and develop outside the interests of a single developer. It also eliminates the danger of the software becoming an abandonware. It should be noted here though that Impromptu is a very powerful tool for audio-visual composition and it would be difficult to find areas that have not been addressed by the author.

### 5.2. Learning Impromptu

The biggest challenge for me when aiming to write music in Impromptu was to learn Scheme. The language paradigm was foreign to me, even if SuperCollider affords functional programming. It involved me ceasing to think of musical data as something one stores in objects with state variables, parameters and methods, and instead think of functions that parse data in a stateless manner. Functional languages frown upon state variables that can be overwritten through time, and this arguably results in fewer bugs [13]. This also means that instead of for-loops, one has recursion. Starting to think in recursion and the lack of objects was for me the hardest nut to crack in learning Scheme. It drastically changed my way of thinking programming, or rather added to the array of techniques, and I am thoroughly enjoying that experience. One is reminded of Perlis' quote "A language that doesn't affect the way you think about programming, is not worth knowing" [30].

After some considerable frustrations with the unfamiliar nature of functional programming as written in Scheme, I slowly began to appreciate the language. Functions can be written and redefined in real-time, allowing one to redefine parts of the program in the middle of a performance. This is a liberation from the chores of many object orientated languages, where classes have to be recompiled for every change made. Furthermore, by applying macros one can use Scheme as a meta-language in which one can build one's own language structures. A user that works in Scheme for some time invariably will have built his/her own libraries or even sublanguages for the tasks specific to the user. For the composer with highly idiosyncratic

[2] The statement that live coding does not involve embodied performance patterns can be questioned. In a personal communication with Sorensen, he says: "the act of learning to quickly touch type define, and finding the bracket keys quickly, and learning to tab complete etc. are all embodied. Indeed I would argue that for much of my live coding these days many of the structures I build are embodied in the sense that I think to myself "I need to make a minor 7th chord here" and my hands just make it happen - in other words I don't really have to *think* about it - it's "under the fingers" in jazz speak."

needs, Scheme is an ideal language. This particular nature of Lisp, or Lisp-based languages such as Scheme, is identified and discussed thoroughly in Taube's key work on computational music [40].

In terms of musical timing Impromptu performs well. Functions are scheduled in time through a solid callback system and will have calculated code before it is needed. From my tests, Impromptu is not as fast as Python or SuperCollider, but timing has never proved to be a problem. The audio is calculated in a different thread as Impromptu is an Audio Unit host, which means that one does not have the same control from the language over the audio synthesis. However, the latest release of Impromptu has implemented a LLVM [19] compiler which allows one to write computationally heavy code, such as audio synthesis, through the definec function [37]. This is equivalent to writing SuperCollider UGens or Max/MSP externals on the fly. I have yet to properly explore the power of this recent addition, although I successfully managed to write a square wave in a performance.

Having used Impromptu for some time, I began to perceive the main difference between object oriented programming and functional programming as being a metaphysical one, i.e., whereas the OOP approach is Platonic in that there are objects, prototypes, properties and methods, the functional approach is Heracleitean, emphasizing flow, process, and the lack of objects with inherent properties. This is manifested in multiple ways, for example, in how I would write dynamic functions to populate lists with note values and recursively through other functions, empty those lists during playing, until they needed populating again. There was never a static entity one could denote as the piece's "melody."

### 5.3. Musicking with Impromptu

As an Audio Unit host, Impromptu is set up such that Scheme functions interact with the AUs. Any published synth parameter of the AU can be controlled from Impromptu, but in practice I typically find myself controlling the synths from the note level, i.e., by sending MIDI note values to the synth. It has forced me to work more at an intermediate level than I'm used to, the level between synthesis and the meta levels of generative composition or application development. I imagine that this is liberating to many composers since, instead of a terminology typically characterised with words such as "frequency," "amplitude," "envelopes" and such, one is operating with "notes", "scales", "beats", "bars", and "metronomes". Of course one cannot generalise here, since the Pattern system in SuperCollider can be very high level and one can also choose to work at the synthesis level in Impromptu. However, there is a clear difference in emphasis, deriving equally from the language foundations and the culture around the environment. I speak as a SuperCollider user, but I do understand the background to this: SuperCollider derives from the field of audio synthesis – it was initially designed as a synthesizer that could be algorithmically controlled – whereas Impromptu, through languages such as Lisp and Scheme, traces its origins more from the field of AI, as represented by Taube's book [40].

These two distinct origins, I perceive, result in distinct musical practice: whereas SuperCollider users focus largely on synthesis, signal processing, and generative audio, Impromptu users operate more on the more traditional compositional level. This is also manifested in the way each environment is presented in its initiatory literature: whereas the SuperCollider student starts with synthesis and might end up using the more musical Pattern libraries, the student of Impromptu begins with writing note-level compositions and perhaps going into synthesis from there. This can be studied with evidence on a recent Computer Music Journal DVD on live coding systems [28] where it is clear how SuperCollider and Impromptu users differ with regards to levels of operation.

Personally, I have enjoyed working at the "metalevel" and Impromptu has profoundly inspired my thinking and compositional approach. However, I could not get accustomed to the use of Audio Units: I found the sounds often too synthetic, stale and lacking life. Moreover, the lack of control, understanding, and design of their parameters frustrated me. I realised that I much prefer to understand my sound sources perfectly to the minor details, even if the sound might be less sophisticated than achievable with an Audio Unit synth. I therefore decided to write an Impromptu client for the SC Server that would adhere perfectly to the way Impromptu works, but one would be controlling synths, groups, nodes and busses on the SuperCollider server. This client is called SCIMP.

## 6. SCIMP

The SuperCollider Server is designed with the aim of being a highly effective and streamline synthesis engine with a simple interface controlled with Open Sound Control (OSC). The idea was to separate the composition language from the synthesis engine [25]. The synth can therefore be controlled from any software that supports the OSC protocol, whether it is SuperCollider, Java or indeed some specific hardware.

There already exist various types of SuperCollider clients [32]. Most of them have made an effort to make abstractions of Synths, Nodes, Groups and Busses, as modelled in the object orientated SuperCollider language. Initially this seemed to me to be the most natural way to proceed. However, since one does not really write classes in Scheme and there is no inherent object orientated system, this proved to be strenuous. Of course, the flexibility of Scheme allows one to create an object orientated system if required. After a discussion on the mailing list, Sorensen posted such a system and I considered using that in my client. Having implemented various tests, it slowly dawned upon me that my Impromptu SC client should rather conform to the functional design philosophy of Scheme and the way the Impromptu play-note function works. Having designed small functions that represent each of the commands the

SuperCollider synth accepts, composition with Impromtu using SC Server as the synthesis engine turned out to be relatively simple and mirrors well the way one works with Audio Units.

```
;; example using group

;; Working with the Node system of SC Synth in Impromptu
(define synth3 (next-node-id))
(sc:synth:new (now) synth3 "scimp_buf" 0 0 "bufnum" mybuf "loop" 1)
(define delay (next-node-id))
(sc:synth:new (now) delay "scimp_delay" 0 0 "in" 10 "out" 0)
(sc:node:set (now) synth3 "out" 10)
(sc:node:after (now) delay synth3)
(sc:node:set (now) synth3 "out" 0)
(sc:synth:free (now) synth3)

;; Playing a little melody through a delay
(define foo
   (lambda (out)
      (sc:synth:grain (now) "marimba" "freq" (random 333 589) "amp" 1 "out" out)
      (callback (+ (now) (random '(10000 15000))) 'foo out)))

(define delayplay
   (lambda ()
      (define delay (next-node-id))
      (sc:synth:new (now) delay "scimp_delay" 0 0 "in" 10 "out" 0)
      (foo 10)))

(delayplay)
(define foo '())
```

**Figure 3**. Scimp example. This code shows how a node is created and used to create a synth that is passed through an effect synth.

The Scimp client is therefore different to common client design for SC Server, where complex Synth, Node, Group and Bus classes are created. In Scimp, the only state variables stored on the Impromptu side are node numbers, i.e., the reference to the synth or the group on the server UGen graph tree.[3]

## 7. THE IXI LANG MATRIX

Working with Impromptu for some months has changed the way I think about programming and how I solve computational or musical problems. Gradually, a new metaphorical landscape presented itself. An example of this influence on my thinking can be found in a recent addition to the ixi lang [22], the *matrix*, which is directly inspired by functional programming. The matrix can be called up from ixi lang, accessing the same instruments and effects. It is simply a matrix with rows and columns where each of the elements store instructions and code. Each vector is given a direction, speed, instrument, note, and maybe some SuperCollider code interpret, thus giving ixi lang access to the much more expressive SuperCollider language.

As each cell of the matrix is effectively a vector with a direction and speed, the matrix has to be populated with actors that move through the matrix and play the instruments or run the code. These actors run from the default tempo clock of the ixi lang, and stay temporally in sync with all tempo changes in other ixi lang scores and between different matrices. Storing code in such vectors is a design feature inspired by functional programming, where programming structures can be represented as nodes in a network of dynamic flows. This enables quick design of code that can be included in an already running program, or musical score, in a temporally sound way by inheriting the ixi lang's default tempo clock. All these features are inspired by Impromptu in some way or another.

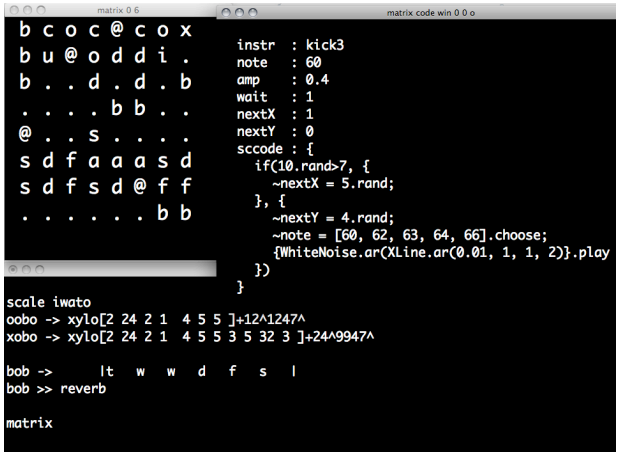


**Figure 4**. A screenshot of the ixi lang matrix. The matrix of the left consists of characters that contain instruments and SuperCollider code. Agents (@) run through the matrix and trigger code stored in the cells.

## 8. EVALUATING CODING SYSTEMS

For many of the practitioners of live coding, it is compelling to frame the compositional process as an improvisation by revealing to the audience not only their musical, programming, and typing skills, but also thought patterns. This poses the question of the similarity between computer games and music, perhaps with the general distinction that the former tend to have winning as a goal, therefore focusing on the end, whereas the latter embraces collaboration, emphasizing the process. Regardless, we are immediately presented with the etymology where people "play games" and "play music," where these activities take place in a medium famous for blurring most traditional distinctions in music.

Live coding as an electrifying performative and improvisational act is in more than one way related to the excitement of playing computer games. Firstly, there is a shared underlying thinking in terms of software design and the inclusion of objects that are capable of changing state. Secondly, the conceptual and visual metaphors in live coding are often borrowed from computer games. Finally, the concepts of gameplay [9] and playability [33] are important in live coding: some of the measurement criteria of playability apply strongly, e.g., satisfaction, learning, efficiency, emotion, immersion and so on. Live coding thus sits solidly at the intersections of music, performance, computer science, and games, and should be experienced and evaluated as such.

With the ever increasing flora of live coding environments available [28], it is timely to investigate the usability of live coding languages and explore their design from a HCI and game design point of view. There are good heuristic measurements available [8, 15]

[3] It should be noted that Rohan Drape has written a Scheme client for SC Server (www.slavepianos.org/rd/sw/rsc3/), but it does not work within Impromptu and it presents a slightly different design ideology.

but what might possibly come out of such research is that every live coding system is highly idiosyncratic and that it is difficult and complicated to compare the users. Live coding systems often require strong programming skills of diverse programming paradigms (e.g., imperative or functional programming) making neutral user testing, in the form of game testing or HCI usability studies, very difficult. The approach proposed in this paper is to engage with this problem by applying selected qualitative research methodologies.

## 9. CONCLUSION

In the introduction I stated that this paper is a progress report. One never finishes learning a language and the topic of technological conditioning is a key research interest of mine. Therefore, more observations are due to follow from this research project where I will report on studies of other people's learning processes. However, this paper has elucidated how a specific programming language defines the musical thinking of a composer and changes the ways of thinking through the habituation of learning it. This is evident when I have gone back to working in object oriented languages. I have started to write code that exhibits patterns derived from functional languages and I realise in many occasions that I think differently about software design.

This paper discussed how it is not only the language that affects the user's musical creativity, but the culture around it as well. A comparative study would be interesting in this context, since here the focus was on Impromptu. Even if it is difficult to find a musical style that is common with the users of each environment, there are strong practices of coding that influence the way people conceive of their work. As an example, Impromptu has much stronger focus on live coding as a musical practice than one finds in SuperCollider. An informal study shows that there is an unusually high percentage of Impromptu users who live code.

This project has been musically inspiring, as it has forced me to strengthen the mental compartmentalisation of the note and the synthesis level. Having a note level control in Impromptu, and a sound engine either as Audio Units or as SC Server, represents a strong separation between what in Max Matthews' MusicN systems was called the score and the orchestra. SuperCollider 3 blurs this distinction in many ways, although not as perfectly as one found in SuperCollider 2. Having said that, with the recent definec function in Impromptu, one can write DSP code in realtime through the JIT compilation into LLVM. I have yet not explored this interesting feature fully, but it provides a further effacement of the artificial distinction between musical events and synthesis [37].

In the introduction I described the difficulties, or the near impossibility, of comparatively studying live coding environments due to the markedly different background of the participants and the diverse musical goals of the live coders themselves. However, we can learn much from theorists in computer games and human-computer interaction. Studies and surveys can be performed, although strong quantifiable results should not be expected. This research project aims at gaining an understanding of how people engage with these "machines for thinking" through teaching, giving workshops, surveys, and importantly, by acknowledging what a first person reflective account can give in terms of valuable data for analysis and interpretation.

## 10. REFERENCES


[1] Anderson, B. G. *Around the World in 30 Years: Life as a Cultural Anthropologist*. Waveland Press, Long Grove, IL, 1999.

[2] Azoulay, K. B. *Black, Jewish, and interracial: it's not the color of your skin, but the race of your kin : and other myths of identity*. Duke University Press, Durham, 1997.

[3] Bartleet, B-L. & Ellis, C. (eds.) *Music Autoethnographies: Making Autoethnography Sing/Making Music Personal*. Australian Academic Press, Bowen Hills, Qlnd, 2009.

[4] Brown, A. R. & Sorensen, A. "Interacting with Generative Music through Live Coding", *Contemporary Music Review*. Vol. 28 (1). 2009, pp. 17–29.

[5] Collins, N. "Live coding of Consequence", *Leonardo Journal*. Vol. 44 (3). 2011.

[6] Collins, N., McLean, A., Rohrhuber, J. & Ward, A. "Live Coding Techniques for Laptop Performance", *Organised Sound*, vol. 8 (3). 2003. pp. 321–30.

[7] Dennett, D. *Against Method: Outline of an Anarchistic Theory of Knowledge*, University of Minnesota Press, Minneapolis, 1970.

[8] Desurvire, H., Caplan, M. & Toth, J. A. "Using heuristics to evaluate the playability of games", *Proceedings of CHI '04*. ACM, New York, 2004.

[9] Ermi, L. and Mäyrä, F. "Fundamental components of the gameplay experience: Analysing immersion", S. de Castell & J. Jenson (eds.), *Changing Views: Worlds in Play. DiGRA Second International Conference*, 2005.

[10] Feyerabend, P. *Against Method*. New Left Books. 1975.

[11] Goriunova, O. & Shulgin, A. *read_me - Software Art and Cultures*. Aarhus University Press. 2004.

[12] Heidelberger, C. A. and Uecker, T. W. "Scholarly Personal Narrative as Information Systems Research Methodology", *MWAIS 2009 Proceedings*. 2009.

[13] Hughes, J. "Why Functional Programming Matters", *Research Topics in Functional Programming*. ed. D. Turner, Addison-Wesley, 1990, pp 17–42.

[14] Ihde, D. *Technology and the Lifeworld: From Garden to Earth*. Bloomington: Indiana University Press. 1990.

[15] Korhonen H., Paavilainen J., Saarenpää H., "Expert Review Method in Game Evaluations - Comparison of Two Playability Heuristic Sets", *Academic MindTrek 2009 Conference*, ACM Press. 2009. pp. 74-81.

[16] Kuhn, T. "Objectivity, Value Judgment, and Theory Choice", *The Essential Tension: Selected Studies in Scientific Tradition and Change*. University of Chicago Press. 1977.

[17] Kübler-Ross, E. *The Wheel of Life: A Memoir of Living and Dying*. Simon & Schuster, 1998.

[18] Latour, B. *Pandora's Hope: Essays on the Reality of Science Studies*. Harvard University Press. 1999. p. 178.

[19] Lattner, C. & Vikram A., "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation", *International Symposium on Code Generation and Optimization*. 2004.

[20] Magnusson, T. "Expression and Time: The Question of Strata and Time Management in Art Practices using Technology", *The FLOSS + Art Book,* Poitiers: goto10, 2009.

[21] Magnusson, T. "The Acoustic, the Digital and the Body: A Survey on Musical Instruments", *The NIME 2007 Conference Proceedings*. New York: New York University. 2007.

[22] Magnusson, T. "The ixi lang: A SuperCollider Parasite for Live Coding", *Proceedings of ICMC*, 2011.

[23] Magnusson, T. *Improgramming:* http://improgramming.wordpress.com/

[24] McCartney, J. "A Few Quick Notes on Opportunities and Pitfalls of the Application of Computers in Art and Music", *Proceedings of Ars Electronica*, 2003.

[25] McCartney, J. "Rethinking the Computer Music Language: SuperCollider", *Computer Music Journal*, vol. 26 (4), 2002. pp. 61 – 68.

[26] McLean, A. & Wiggins, G. "Texture: Visual Notation for Live Coding of Pattern", *Proceedings of ICMC*, 2011.

[27] McLean, A., Griffiths, D., Collins, N & Wiggins, G. "Visualisation of Live Code", *Proceedings of Electronic Visualisation and the Arts Conference*, London. 2010.

[28] McLean, A., Magnusson, T. & Collins, N. Live Coding DVD supplement. *Computer Music Journal*, vol. 35 (4). 2011. Forthcoming.

[29] Nilson, C. "Live Coding Practice", *The Proceedings of NIME*. 2007.

[30] Perlis, A. "Epigrams on Programming", *Sigplan Notices*. Vol. 17 (9), 1982. pp. 7 – 13.

[31] Polanyi, M. *The Tacit Dimension*. Garden City, New York: Anchor Books, 1967.

[32] Rutz, H. H. "Rethinking the SuperCollider Client…", *Proceedings of the SuperCollider 2010 Symposium*, Berlin, Germany. 2010.

[33] Sánchez, J. L. G., Zea, N. P., Gutiérrez, F. L., Cabrera, M. J. & Rodríguez, P. P. "Playability: The Secret of the Educational Videogame Design", *Proceedings of the 2nd European Conference on Games-Based Learning*. Barcelona, Spain, 2008.

[34] Sorensen, A. http://lists.moso.com.au/pipermail /impromptu/2011-January/000789.html

[35] Sorensen, A. "Impromptu : an interactive programming environment for composition and performance", *Proceedings of the Australasian Computer Music Conference*. 2005.

[36] Sorensen, A & Brown, A. "aa-cell in Practice: An Approach to Musical Live Coding", *Proceedings of ICMC*, 2007.

[37] Sorensen, A. & Gardner, H. "Programming with time: cyber-physical programming with impromptu", *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. 2010.

[38] Stiegler, B. *Technics and Time, 1: The Fault of Epimetheus*. Stanford: Meridian. Stanford University Press. 1998.

[39] Sudnow, D. *Ways of the Hands*. Cambridge, MA: MIT Press. 2001.

[40] Taube, H. *Notes from the Metalevel*. Taylor & Francis Group, London, 2004.

[41] Varela, F. J. and Shear, J. "First-person Methodologies: What, Why, How?", *Journal of Consciousness Studies*, vol. 6 (2–3), 1999, pp. 1–14.

[42] Wadler, P. "Functional Programming: Why no one uses functional languages", *SIGPLAN Notices*. Vol. 33 (8) pp. 23-27, 1998.

[43] Wittgenstein, L. *Philosophical Investigations*. Oxford: Blackwell, 1994.