Embodiment of code

Marije Baalman
nescivi
marije@nescivi.nl

ABSTRACT

The code we read on the screens of a livecoding performance is an expression of our compositional concepts. In this paper I reflect on the practice of livecoding from the concept of embodiment as proposed by Varela, Thompson, and Rosch (1991): how we as coders embody the code, how machines can embody code, and how we inevitably need to deal with the physicality of the machines and our own bodies that we use in our performances, and how we adapt both our machines and bodies to enable our performances.

1. Introduction

This paper is an expansion of the author's thoughts on the topic of embodiment in the context of livecoding, following the invited talk at the *Livecoding and the Body* symposium in July 2014. As such it is a starting point for further research and reflection on the practice of livecoding and the role of our bodies in it.

Varela, Thompson, and Rosch (1991) defined *embodied* as follows (pages 172-173):

> "By using the term embodied we mean to highlight two points: first that cognition depends upon the kinds of experience that come from having a body with various sensorimotor capacities, and second, that these individual sensorimotor capacities are themselves embedded in a more encompassing biological, psychological and cultural context."

In the context of livecoding as a performance practice, this concept of embodiment surfaces in various ways - the physical interaction of our body (sensorimotor interactions of typing, viewing what the screen shows us) with the machine (its interface to receive input and giving output, but also the hardware architecture that defines the body on which code is interpreted), as well as how the grammar and structure of a programming language shapes our thinking about concepts and algorithms, and in turn the development of a programming language is determined both by the hardware on which it will run, as by our desire to express certain concepts. Similarly, Hayles (2012) writes in her book *"How we think"*:

> "Conceptualization is intimately tied in with implementation, design decisions often have theoretical consequences, algorithms embody reasoning, and navigation carries interpretive weight, so the humanities scholar, graphic designer, and programmer work best when they are in continuous and respectful communication with one another."

Where she suggests a close collaboration between humanities scholar, designer and programmer - within the livecoding scene, many performers write (and share) their own tools, i.e. they are bridging the artistic and the technological within their own practice, and actively contribute to the context within which they work; shaping their enviromenment at the same time as they are navigating through it. This refers back to both the second part of the definition of *embodied* of Varela, Thompson, and Rosch (1991).

In this paper I will first reflect on how the properties of a programming language influences our thinking and expression in code, then I will look at how our computers can embody different aspects of code, before discussing the physicality of the machine and how our code not only has its intended effects on the output media, but also on the machine itself. In the last section, I go into the interaction loop between the coder and the machine, and how we attempt to optimize this interaction loop.

2. Code embodied by the human

The practice of livecoding music has connections to both algorithmic music, and its expression in computer music, as well as to what in dance is termed *instant composition*: (Collective 2015):

> "*INSTANT COMPOSITION* combines the notion of working from the moment (INSTANTaneous creation) with the intention to build something (COMPOSING a piece with an audience present). This means that for us, improvisation principles are always concerned with both the question of FREEDOM and the question of STRUCTURE."

We express our concepts for musical structure in code - and as livecoders we do this at the moment of performance, rather than completely in advance - we change the algorithm, while it is unfolding its structure. Where in improvisational dance or improvised music, the structure of the composition is not apparent to the viewer or listener - in livecoding - when the screen is projected for the audience to see - the improvised compositional structure is made apparent, although it may be changed again by the livecoder before it is actually heard.

In our first encounters with a programming language as a coder, we try to adapt our minds to understand the language and develop our own methods and 'ways of doing things' within the language. That is, we create our own subset of the language that we use in our daily conversations. If we find we are dissatisfied with what we can express, we either look for another existing programming language that fits our needs, or we write our own (extensions of a) language to enable the expressions that we want. At the same time, the longer we spend programming in a particular language, the more our mind gets attuned to the language and shapes our thinking about concepts to express. Not only our mind gets attuned, also our body - as we type certain class names, our fingers can become attuned to particular sequences of letters that form words that have meaning in the code. As such code is an embodiment of our concepts (Varela, Thompson, and Rosch 1991), which develops itself in a dialogue between ourselves and the coding language.

(Rohrhuber, Hall, and De Campo 2011) argue that adaptations of a language (in their discussion the "systems within systems" of SuperCollider) are "less like engineering tasks than works of literature, formal science, and conceptual and performative art", and they place this practice within a social context of sharing code and such adaptations with other coders. Thus, code is not only the embodiment of concepts in our own mind, but they co-evolve with concepts of others, which feed into new adaptations of code and programming languages, which may lead to new concepts to express, or musical composition to be made.

On a social scale, in a comment made in a conversation at the SuperCollider Symposium in London in 2012, Scott Wilson commented how he enjoys the community around the programming language, in that even though the composers/musicians within the community have vastly different styles of music they create, they share a common mindset. This may well be because either the programming language attracts people with a certain mindset, or using the programming language pushes towards a certain way of thinking - or perhaps a combination of these.

3. Code embodied by the machine

In his description of the B1700 architecture, Wilner (1972) starts out describing how programmers using Von Neumann-derived machines are forced 'to lie on many procrustean beds'. He positions the motivation for the architecture of the Burroughs computers against the rigidity of the Von Neumann-derived architecture. In contrast the Burroughs machines are designed based on the requirements of the programming languages that should run on them, and the kinds of tasks that these programs need to perform. This resulted in features of the hardware that even today are still unheard of, such as variable bit-depth addressable memory and user-definable instructions in the instruction set of the B6000 machines (Mayer 1982). While in their first machines, the architecture was designed specifically for the programming languages ALGOL and COBOL, in the design for the B1700 they designed an architecture that would be able to adapt to a variety of programming languages through the use of emulators that they called *S-machines* which could interpret application programs. The hardware of the Burroughs machines was more complex than the contemporary machines of competitors, but it embodied many routines that would otherwise have needed to be handled by software routines (Mayer 1982). It made for a very close relationship between the code written in a high level language and the resulting machine code, considerably reducing the time to design and run these programs. Far ahead of their times, the Burroughs machines are an outstanding example of the embodiment of code on the side of the machine.

For specific purposes, such as graphics calculations or digital signal processing (DSP), there are specialised computation units that have been designed with these applications in mind: these DSP CPU's have an architecture that allows them

to do the same mathematical operation on a vector of data points at the same time, thus speeding up computations on large data sets.

At the other end, for the Von Neumann derived architectures, programming languages have been "stretched on procrustean beds", that is they have been optimized to run on the hardware architecture, favouring specific methods that fit the architecture (e.g. byte-oriented coding, the development from 8-bit, to 16-bit, to 32-bit to 64-bit machines) over ones that fit the application (e.g. working within the decimal system like one of the Burroughs machines (Mayer 1982)). In those parts where code needs to be optimised to run fast on specific hardware, certain programming strategies are preferred over others, and have shaped common practices of coding based on the hardware properties.

Most modern programming languages, and notably the programming languages used by livecoders, are further and further away from the hardware level; given the fast processors and the generous memory space we have today, speed and memory constraints are less of an issue in the design of programming languages, and the languages can be geared more towards particular ways of expressing algorithms. The interpreter or compiler takes care of translating these into machine code to run on the hardware, and in most cases the programmer in a high level language does not need to be aware of the underlying hardware.

4. The physicality of the machine

Having built-in the possibility of extending the instruction set, the designers of Burroughs were at the same time aware of security risks, so they built in a flag into the storage of data that would determine whether the word could change the instruction set or not; only programs that were part of the operating system were allowed to make this change (Mayer 1982). Also with the Von Neumann-derived machines, the now common (modified) Harvard architecture (Wikipedia 2015), makes livecoding — seen from the machine's point of view — impossible: we cannot change the program instructions at the very momemt they are running. We can just manipulate code that is compiled on-the-fly or interpreted into a sequence of machine instructions that the machine knows. But once it is in the pipeline towards the CPU (or in its cache), we cannot interfer anymore. We cannot change the machine, we can just change the commands that we put into the cache. Only if we interfer with the hardware of the machine itself, we can make a change. In Jonathan Reus' *"iMac Music"*, this is exactly what he is doing, while the machine is executing software, the performer hacks into the circuitry, changing the pathways of the electronics and distorting the images and sounds created by the iMacs (Reus 2012).

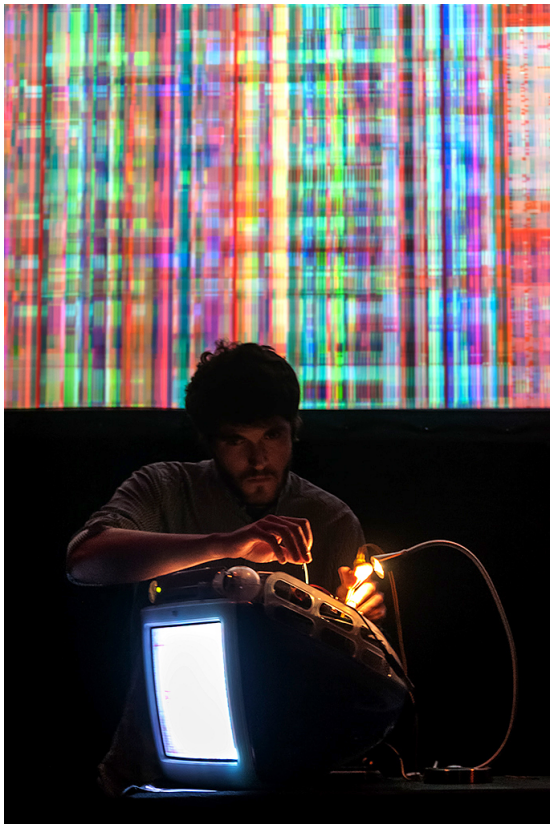

Figure 1: *Jonathan Reus performing "iMac music", photo by Ed Jansen.*

The instruction sets of computers consists very basic elements, methods of fetching data from memory, manipulating these bytes of data (negation, bit shifting, etc) and combining of data (addition, subtraction, multiplication, etc). What we call code is for the machine just data, through which we manipulate the memory map that determines how we move through this data. The compiler or interpreter translates our data to machine instructions that will determine these movements.

Griffiths (2010) with his project *"BetaBlocker"* created a virtual "8 bit processor with 256 bytes of memory" that can only play music, with a small instruction set. In his perfomances the state of the machine is visualised, as you listen to the sound that is created; in that *"BetaBlocker"* exposes the (virtual) machine completely, and the livecoding is happening close to the physicality of the machine.

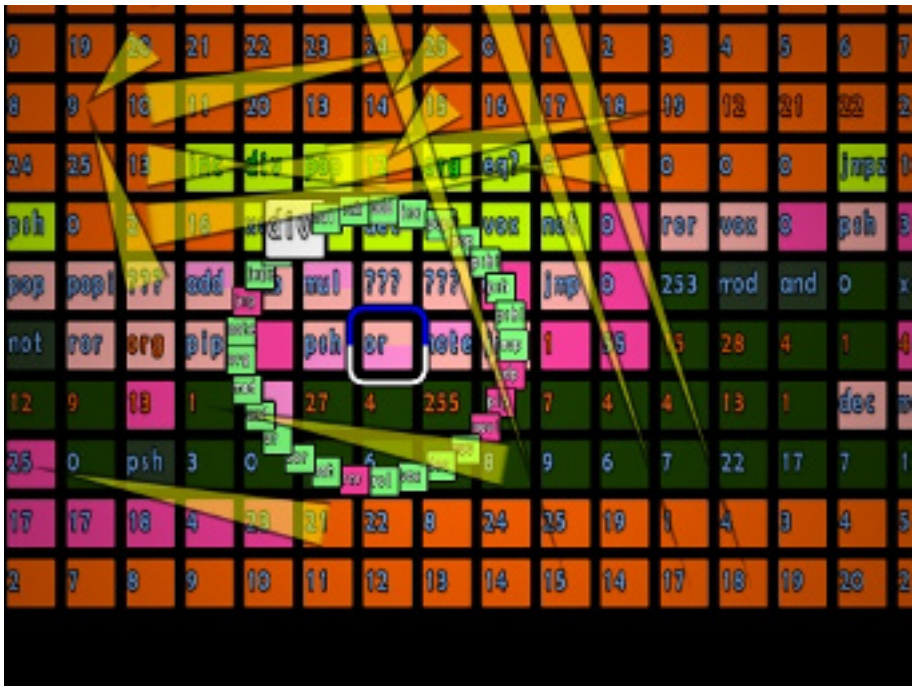

Figure 2: *Dave Griffiths' BetaBlocker (image from McLean et al. (2010)).*

The data that flows within the computer can come from and go to different sources; the data is communicated to the CPU through various buses. The processing of this data has both intended and unintended effects. If we write an algorithm for sound synthesis, the data we process is intended to go to the audio card and be transformed into sound waves. But at the same time when the CPU is very busy, it heats up, and safety mechanisms are in place to ensure that as soon as a high temperature is measured, the fan within the computer turns itself on. Sometimes the effect can be both intended and unintended, e.g. when accessing the harddisk (we intend to get data from the harddisk, but we do not necessarily intend it to go spinning), or refreshing the screen (we want to see an updated text, but in order to do so we regularly refresh the screen). Both of these things cause alternating electromagnetic fields around our computers, which we do not necessarily intend.

Following Nicholas Collins' circuit sniffing method (N. Collins 2006) using electromagnetic pickups to look for the sound inside the machine, (Reus 2011) is transforming these sounds and manipulates them with code, and thus causing new sounds to be created by the machine, in his work *"Laptop Music"*.

In both *"iMac Music"* and *"Laptop Music"*, Reus draws attention to the physicality of the machine, and how the code that runs on it can be affected by hacking into the hardware, or how the code running on the machine affects what happens inside the body of the machine. Both works remind us of the limits of the machine and the processes that happen inside it - how the machine has a life of its own. For code to come to life — to express its meaning — it needs a body to do so. Code needs to be embodied by a machine and its meaning cannot manifest without this machine.

5. The human body and the machine

When we livecode, we try to translate the concepts in our minds to code that runs on the computer, that will result in some audible or visible or otherwise experienceable outcome. Let us look at this in detail (see also figure 3): in our mind

we formulate concepts into code words that will make up the program, we then send commands through our nervous system to control our arms, hands and fingers to type in these words on the keyboard. The keyboard sends signals to a bus, the CPU gets a notification of these signals, and sends them to our editor program, which displays the code on the screen, and at the same time, puts the code words into the computers memory, so that given the right command, the editor can send the code to the interpreter, which will translate the code into machine code that is processed by the CPU to computer output data, which is then sent to output media, such as sound, light or visuals. Both the information from the display screen of the computer (our view on the editor), and the result of the code as observed from the output media, drive us to adjust our motor output (correct mistakes in our typing of the code, as we see it happen), as well as adapt our concepts or come up with the next ones, as we see the output media going into a certain direction.

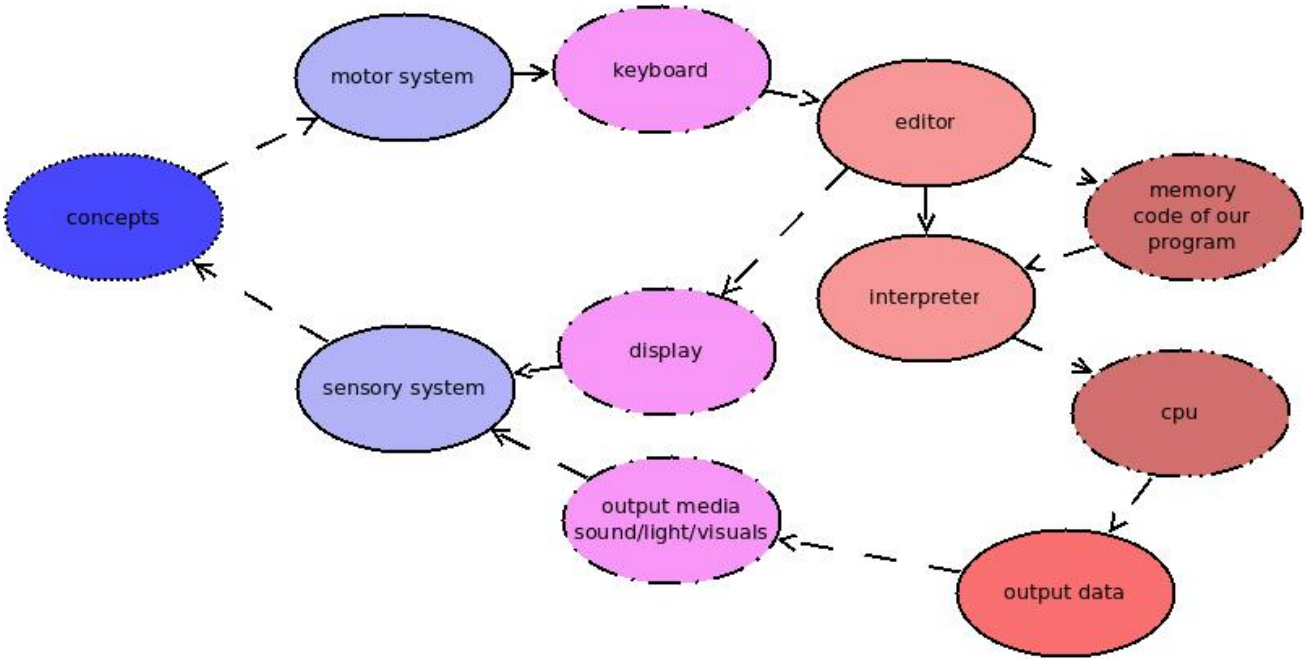

Figure 3: *Schematic view of interaction of the human body interfacing with the body of the machine to translate our concepts into code.*

As we prepare for the act of livecoding, we try to optimize the qualities of the machine - both the physical and the software environments within which we code - to make this interaction loop go faster and smoother; we may choose a specific keyboard and a screen, we choose our editors and how they display code, and we choose our output media. At the same time we train our bodies to perform the motor skills to control the machine better, such as learning how to touch type, fast switching between mouse and keyboard, using keyboard shortcuts, and so on. We train the motor skills needed for typing and correcting mistakes, so that they take less of our brain time to think about concepts (when I type, I tend to feel when I make a mistake, sometimes even before I see the letters appear on the screen; before I consciously realise I made a mistake, I already hit the backspace key to go back and correct the mistake). We have a need for speed, we want to prevent the bodies of the human and the machine from obstructing the translation of concepts to code that the machine understands. We do this by adapting our bodies; the physicality of the machine is important (the keyboard layout, the feel of the keys, the feel of the mouse or touchpad), and we even embody part of the coding languages in our body; if you always use sine oscillators (*SinOsc*) and never saw tooths (*Saw*), your fingers will almost automatically type *Si*, before you can hold back and adjust to type *Sa* in the rare occasion that you want a different sound.

Similar to how we put the code into the computer's memory, we keep in our own memory a representation of the code. When I work on code, I usually have a map in my mind of the structure of the code - when an error occurs, I first visualise internally where the origin of the error might be. I tend to object to the notion that SuperCollider (the object-oriented language I usually work with) is unlike the *visual* programming environments (such as PureData and Max/MSP), because its editor interface is text-based. In my mind, I navigate the code I work on as a visual map of interconnected objects.

In the performance *"Code LiveCode Live"* (Baalman 2009), I *"livecode the manipulation of the sound of livecoding, causing side effects, which are live coded to manipulate the sound of livecoding, causing side effects, ..."*. The concept of this livecoding performance is to only use the sound of the typing on the keyboard of the laptop as the source material for the audio in the performance. Furthermore, the sensing devices embedded in the laptop (e.g. accelerometers, touchpad, trackpoint, camera, the keyboard) are made accessible and mapped (through livecoding) to the manipulation of the sound material. Also the processes of the code itself (e.g. their memory and cpu usage) are used as sensors and used in the performance.

As such the performance of livecoding is influenced through its own side effects, transforming the code not only in the logical, programmatical manner, but as an embodied interface of the machine on our lap.

On the SuperCollider list an interesting comment was made about this performance; a viewer had watched the online documentation of the piece, and commented in a thread about the auto-completion features of the SuperCollider-Emacs interface that it seemed rather slow. I had to write back that no auto-completion features of the editor were used in that particular screencast. Apparently I succeeded in embodying the SuperCollider code to such an extend, that it seemed an enhancement of the editor, rather than the training of my body.

6. Conclusion

In this paper, I have reflected on different aspects how embodiment takes place within the livecoding practice. I emphasize on the aspects of the physicality of the act of livecoding - livecoding is not just the development of algorithms over time, our bodies play an important role in the performance practice of livecoding; likewise, the physicality of the machine plays an important role - its architecture determines in the end what the outcomes can be. We can only speculate what livecoding would have looked like today, if the Burroughs machine architectures had been predominant since their first invention — would we have had hardware that was designed to run SuperCollider so we would hardly need a compiler to run the code? Can we imagine a different interface than the keyboard and the screen to input the code into the machine, that will allow for a different way of embodying the code? What programming languages would evolve from such an interface? What kind of conversations would we have?

7. References

Baalman, Marije. 2009. "Code LiveCode Live." https://www.marijebaalman.eu/?cat=17.

Collective, Carpet. 2015. "Interdisciplinary Instant Composition - to Define, Share and Develop Concepts of Improvisation." http://www.instantcomposition.com/.

Collins, Nicolas. 2006. *Handmade Electronic Music : the Art of Hardware Hacking*. Routledge, Taylor; Francis Group.

Griffiths, Dave. 2010. "BetaBlocker." https://github.com/nebogeo/betablocker-ds.

Hayles, N.K. 2012. *How We Think: Digital Media and Contemporary Technogenesis*. University of Chicago Press. http://books.google.nl/books?id=737PygAACAAJ.

Mayer, Alastair J.W. 1982. "The Architecture of the Burroughs B5000 - 20 Years Later and Still Ahead of the Times?" *ACM SIGARCH Computer Architecture News* 10 (4): 3–10. http://www.smecc.org/The%20Architecture%20%20of%20the%20Burroughs%20B-5000.htm.

McLean, Alex, Dave Griffiths, Collins Nick, and Geraint Wiggins. 2010. "Visualisation of Live Code." In *Electronic Visualisation and the Arts (EVA), London 2010*. http://yaxu.org/visualisation-of-live-code/.

Reus, Jonathan. 2011. "Laptop Music." http://www.jonathanreus.com/index.php/project/laptop-music/.

———. 2012. "IMac Music." http://www.jonathanreus.com/index.php/project/imac-music/.

Rohrhuber, Julian, Tom Hall, and Alberto De Campo. 2011. "Dialects, Constraints, and Systems Within Systems." In *The SuperCollider Book*, edited by S. Wilson, D. Cottle, and N. Collins. Cambridge, MA: MIT Press.

Varela, Francisco J., Evan Thompson, and Eleanor Rosch. 1991. *The Embodied Mind: Cognitive Science and Human Experience*. MIT Press.

Wikipedia. 2015. "Harvard Architecture." http://en.wikipedia.org/wiki/Harvard_architecture.

Wilner, Wayne T. 1972. "B1700 Design and Implementation." http://bitsavers.trailing-edge.com/pdf/burroughs/B1700/Wilner_B1700designImp_May72.pdf.