

Patterns of movement in live languages

Alex McLean

Goldsmiths, University of London
ma503am@gold.ac.uk

Geraint Wiggins

Goldsmiths, University of London
g.wiggins@gold.ac.uk

November 11, 2009

Programmers do their work by *writing* – a piece of software is a structure made from words. These structures are generally too big to comprehend in their entirety, so programmers instead focus on small detail and overall plans; zooming in to find parts to combine and simplify and zooming out to find places to build. But this is not architecture: these structures are more like machines than static buildings. A programmer’s work is set in motion by a program interpreter, with information flowing in and around processing units before being directed outward in response.

Usually a programmer will write some text, and then step back to start it up, watch it work and decide upon the next edit. Live coding programmers however work on their software while it is running, as if they were modifying a machine without switching it off first. Because software is built from words, this is done by editing it as text, adding new routines or changing the character of an existing one. Such a change takes immediate effect, allowing fast creative feedback.

Where a written novel exists to describe human activity, written software exists to simulate it. Therefore the live coder can take the role of an artist, constructing simulators in order to generate patterns of movement, either as music, video animation or both. This can be done in front of a live audience, so that the process of writing software becomes the process of improvising music or video in performance art.

Programmers are finally taking to the stage. Introspecting and encoding their musical thoughts before an audience. A tradition of live coding has quickly formed where computer screens are projected, making the programmer’s reactions to their work visible. Questions of authorship disappear; the performance is live, the programmer improvising through the medium of written language.

1 Pre-history of live coding

Words emerge from our thoughts and are modulated into sound using our vocal tract. We perceive words in part as physical movements, embellishing them with prosody even when reading silently from a page. Abstract symbols are grounded in movement.

Through the centuries, musicians have related movements of their instruments to symbolic movements of their vocal tract. The music of the Indian tabla drums is of timbre and not melody, yet shows complexity which according to Patel (2007) would seem impossible for humans to perceive. The musicians and audiences ground their timbre perception using *Bol* syllables, relating articulation of the tabla drum with the articulation of the voice (Patel, 2007, p. 32). Similarly complex systems exist in the tradition of Scottish highland pipers, where *Canntaireachd* vocable words are used to represent articulations of the bagpipes (Chambers, 1980).

Symbolic representation of music then is nothing new, and indeed the interplay between symbols and movements appears fundamental to our experience of music. However, relative to the history of music live coding does appear to be genuinely novel. The earliest known live coding performance was in 1985, by Ron Kuivila at STEIM in Amsterdam (Blackwell and Collins, 2005). Live coding did not get its own cultural identity until TOPLAP, the Temporary Organisation for the Promotion of Live Algorithm Programming was formed at the Changing Grammars workshop

in Hamburg in 2004 (Ward et al., 2004). Although it is possible to do live coding without computers¹, through self-modifying rule-based composition, there is no evidence that this was done before the invention of computers. It would seem that it required the invention of dynamic code interpretation for live coding to appear possible or perhaps desirable.

2 Textual versus visual programming

In human-computer interface design, there has been a progression from text user interface (TUI) to graphical user interface (GUI). Accordingly, a computer culture has developed where text is seen as old fashioned and technical, and graphics are seen as advanced and more human oriented. This progression from computer text to computer graphics goes against the historical flow of technological development, from the 30,000 year old pictographic stone age paintings we see preserved in caves, through ideographic symbols emerging in the 4th millennium BC and to the phonetic and logographic writing systems now in mass use. While comparing graphical and textual systems, we should keep in mind that text is visual, graphical and spatially arranged on the page. To speak of graphics and text in opposition is therefore a category error.

The confusion between textual and visual remains in discussion of live coding interfaces. Languages such as *pure data* and *max* are frequently described as ‘visual’ and therefore more suited to artists than ‘textual’ languages such as *Lisp* and *C*. But what is meant by ‘visual’? If by ‘visual’ we mean ‘graphical’, we have already noted that written language is composed of graphemes. Perhaps by ‘visual’ we mean ‘spatial’, but while *pure data* or *max* allow us to arrange boxes on a page, linguistically speaking, the arrangement is arbitrary, having no syntactic effect or semantic meaning². By this measure, conventional computer languages are *more* spatial: if you change the position of the words on the page then you change the meaning of the program, particularly in languages such as Haskell and Python which have a two dimensional syntax.

The confusion is resolved when we look beyond the syntax and semantics of computer language itself. When programmers write code they are not just communicating with the computer but with themselves and other programmers, adding commentary to ease understanding, and taking care to give names to their abstractions which reflect their purpose. Similarly, a fluent ‘visual’ programmer arranges a patch in shapes meaningless to the language interpreter, but highly meaningful to themselves and perhaps other programmers. These are all *embellishments* ignored by the computer interpreter but of prime importance to human-to-human communication. It is correct then to categorise *pure data* and *max* as primarily *spatially embellished*, and others such as *Lisp* and *C* as *textually embellished* languages.

In computer science, these embellishments are referred to as *secondary syntax*, perhaps a misnomer for syntax that runs parallel to linguistic syntax and is of great importance to human programmers. Indeed, that software interpreters are not concerned with these textual or visual embellishments is a great failing. Why do the textually embellished languages not draw meaning from word morphology? The occasional limited use, such as the exclamation mark in the *ruby* language to effect in-place edits, hints at what could be possible. In electronic music, *vocable synthesis* uses words to describe sounds in a manner analogous to onomatopoeia, translating symbolic sequences into instrumental articulations (McLean and Wiggins, 2008). What would a computer music language based around vocable words look like?

Headway has been made in incorporating spatial and geometrical relationships into computer language. The *reacTable* computer music language has syntax based on the angle and proximity of symbols (Jordà et al., 2007), with the related *TurTan* language applying similar techniques to ‘turtle’ vector drawing (Gallardo et al., 2008). Could this research be taken further towards the compositional abstractions of general computation? Perhaps the constraints and pressures felt by live coders will explore the semantics of morphology of words and shapes further in their design.

¹See http://toplap.org/index.php/Live_Coding_Without_Computers for an informal review.

²apart from execution order in *max*, although relying on this is discouraged in favour of the *[trigger]* object

3 Computational abstractions over time

Text is often taken to be something stable, unchangeable, something that is projected into the future of a reader. [...] But as the activity of programming reveals, neither code nor the deterministic algorithm is created in this way - iteration after iteration the written language shows its influence on the thinking and world of the programmer as much as the code changes with its use.” Rohrhuber et al. (2005)

Live coding environments allow improvisation with computational *abstractions*, building models by defining symbols with other symbols. Instead of working with sound events, the live coder manipulates computations for generating them, in this way working directly with musical structure and style. All musicians are engrossed in musical structure, but with live coding that structure is externalised, made visible as code, and manipulated there.

Through working with structure rather than events, the live coder takes on a different relationship with time. A non-live coder can work in the perceptual present, feeling their way through the music moment by moment. Live coders however work in the future, able to make structure that plays out over the entire rest of their performance (although they may later decide to curtail it). Where the live coder gains in control over the future they lose the sense of being in the present moment. Whereas two instrumentalists are able to perform moment-by-moment interactions in call and response, in contrast some live coding groups such as the celebrated *aa-cell* duo begin their performances by typing in code for several minutes before the first sound is generated.

3.1 Connecting code with the moment

While working in abstractions takes live coders away from the musical moment, some are developing ways of bringing direct physical interaction back into their live coding performances. Dan Stowell clamps a microphone into a harmonica support so that he can type while beat-boxing, using extended vocal technique (Stowell, 2008) while editing Supercollider code to ornament and manipulate his rhythms. Matthew Yee-King performs with electronic drum pads next to his computer keyboard, triggering live coded synthesis as part of the Finn Peters quartet.

Another direction is in developing new live coding languages for more immediate use, with operators and terse syntax optimised for musical expression. The idea-to-code latency can then be reduced to a few seconds, helped further by Interactive Development Environments (IDEs) allowing bugs to be identified and fixed quickly. The Scheme based language environments Impromptu and Fluxus have both shown particularly adept use, the former more for music and the latter more for video.

As live coding progresses towards speed typing of terse algorithms under tense conditions, we should consider whether the resulting stress is healthy. The counter *slow coding movement* has emerged as antidote, its website (<http://www.ludions.com/slowcode/>) calling for the virtuosic element, along with the associated excitement and danger, to be removed from live coding. Slow coders take an alternative approach to connecting with the moment, by relaxing and forgetting about the passage of time altogether. Although this intervention may be tongue-in-cheek, it reminds us that live coders can step away from the rush of modern culture to meditate over slow and considered code edits.

We should also of course consider what the audience think of live coding performances. In live coding performances screens of code are projected to the audience as a gesture of openness, but without really understanding whether the effect on their experience is for good or for ill. As audiences grow, the opportunity to investigate the experience of audience members presents itself, but this research has not yet been done. For now the audience is left to create ways of enjoying live coding performance for themselves.

4 Conclusion

From its heritage of improvised music and computer science, live coding inherits a rich culture of ideas. It would seem that some of these ideas need to be challenged before we can see the full potential of the practice of live coding. Using computer languages for live music and video improvisations puts the focus on the Human-Computer Interaction and Human-Human Interaction mediated by programming languages and their environments. Issues such as human perception, the interplay of symbolic and spatial representations, and the musical consequences of making structure newly explicit come to the fore. This is research based practice, and practice based research in its infancy, and there is much left to be done.

References

- Blackwell, A. and Collins, N. (2005). The programming language as a musical instrument. In *Proceedings of PPIG05*. University of Sussex.
- Chambers, C. K. (1980). *Non-lexical vocables in Scottish traditional music*. PhD thesis, University of Edinburgh.
- Gallardo, D., Julià, C. F., and Jordà, S. (2008). Turtan: a tangible programming language for creative exploration. In *Third annual IEEE international workshop on horizontal human-computer systems (TABLETOP)*.
- Jordà, S., Geiger, G., Alonso, M., and Kaltenbrunner, M. (2007). The reactable: Exploring the synergy between live music performance and tabletop tangible interfaces. In *Proc. Intl. Conf. Tangible and Embedded Interaction (TEI07)*.
- McLean, A. and Wiggins, G. (2008). Vocale synthesis. In *Proceedings of International Computer Music Conference 2008*.
- Patel, A. D. (2007). *Music, Language, and the Brain*. Oxford University Press, USA.
- Rohrhuber, J., de Campo, A., and Wieser, R. (2005). Algorithms today: Notes on language design for just in time programming. In *Proceedings of the 2005 International Computer Music Conference*.
- Stowell, D. (2008). Characteristics of the beatboxing vocal style, tech. rep. c4dm-tr-08-01. Technical report, Queen Mary, University of London.
- Ward, A., Rohrhuber, J., Olofsson, F., McLean, A., Griffiths, D., Collins, N., and Alexander, A. (2004). Live algorithm programming and a temporary organisation for its promotion. In Goriunova, O. and Shulgin, A., editors, *read.me — Software Art and Cultures*.