Visualisation of live code

Alex McLean
Goldsmiths
University of London
alex@slab.org

Dave Griffiths
FoAMvzw
dave@fo.am

Nick Collins
University of Sussex
n.collins@sussex.ac.uk

Geraint Wiggins
Goldsmiths
University of London
g.wiggins@gold.ac.uk

In this paper we outline the issues surrounding live coding which is projected for an audience, and in this context, approaches to code visualisation. This includes natural language parsing techniques, using geometrical properties of space in language semantics, representation of execution flow in live coding environments, code as visual data and computer games as live coding environments. We will also touch on the unifying perceptual basis behind symbols, graphics, movement and sound.

Live coding. Visualisation. Improvisation. Music. Video. Computer game

1. INTRODUCTION

Live coding, the improvisation of video and/or music using computer language, has developed into an active field of research and arts practice over the last decade (Wang and Cook, 2004; Ward et al., 2004; Collins et al., 2003). Live coding is made possible by dynamic language interpreters, which allow algorithms to run while they are being modified, taking on changes without any break in the audio or visual output generated by the code. The development of software becomes part of the art in a very real sense; at the beginning of a typical live coded performance there is no code and no audiovisual output, but the output grows in complexity with the code.

A frequent criticism of computer music is the lack of performance, where an artist hides behind their laptop screen, and the audience is unable to see any activity that might ground their experience of the music (Cascone, 2003). Solutions continue to be explored, with many researchers focusing on developing tangible interfaces which bring the computer closer to a traditional instrument. However, a live coding tradition has developed taking the straightforward approach of projecting whatever is on the artist's screen: the code, moving cursors, the debugging output. The audience is then able to see the human movements and code structures behind an improvisation.

This tradition of projecting screens is itself open to criticism; the audience members may feel distracted, or perhaps even excluded by the projection of code written in language they do not necessarily understand. The alternative of showing nothing, hiding behind a laptop screen, is felt to be untenable, but perhaps more should be understood about the practice of projecting code. Watching the articulations of a live guitarist may enhance the experience of a listener who does not play a musical instrument themselves. Can a live coder elucidate the more abstract thinking gestures of their practice? The search is on for ways of visualising code development that allows non-programmers to enhance their enjoyment and understanding of a live coded piece.

2. PERCEIVING CODE

Generally, a programmer cannot work with their eyes closed; a programmer's text editor is a visual interface.[1] Text editors have gained many features over the last few decades, to the point where we no longer call them text editors but Interactive Development Environments (IDEs). The visual presentation of code has developed its own aesthetic; colour is used to highlight syntax, fonts have been designed for code (e.g. ProFont, proggy), and visual tools for navigating around tree-like code structures. Nonetheless computation is fundamentally about symbol manipulation, and the composition of symbols lies at the heart of every IDE. When our eyes saccade across code, the shapes on the screen are categorised into these symbols, and we perceive them as the tokens (words) and statements (sentences) making up our program. The computer interprets code as a one dimensional string of discrete symbols, but humans perceive it as symbols within a spatial scene. Expert programmers may be able to chunk larger blocks of code as meaningful entities; less experienced live code audiences may become stuck on small details, but an elaborate dance of spatial change to code is evident over time.

Our perception of source code is aided not only by spatial organisation, but also by colour highlighting, in-line documentation and the well-chosen names given to abstractions and data structures. These features are collectively known as secondary syntax[2], being that ignored by the interpreter but of benefit to programmers in understanding and organising their code. A challenge to those pushing the boundaries of programming language design is to find ways of taking what is normally secondary syntax as primary. For example the ColorForth language uses colour as primary syntax, replacing the need for punctuation. Even more radically, the instruction set of the Piet language illustrated in Figure 2 is formed by first order colour relationships within a two dimensional grid; instructions include directional modifiers so that control flow travels in two dimensions. Piet, among many other esoteric languages, is inspired by the two dimensional syntax of Befunge shown in Figure 1, a textual language where arrow-like characters change the direction of control flow. Some languages bordering on mainstream, such as Haskell and to a lesser extent Python, have a syntax that takes two dimensional arrangement into account when grouping statements, although this is otherwise unusual.

Secondary syntax is of great importance to human understanding, despite being ignored by the computer interpreter. Without spatial layout and elements of natural language a program would be next to unreadable by humans. Humans live an embodied existence in a spatial environment, and while we are perfectly able to perform computation, our spatial ability still supports such thought processes (Gärdenfors, 2000). As a result source code, as Human Computer Interface, is a half-way mixture of geometrical relations and symbolic structures. This is true even of the 'patcher' dataflow languages in common use in the digital arts (Puckette, 1988), such as Max and PureData. Patcher languages are often described as 'visual', but in fact all the functions are defined textually, and the visual arrangement is purely secondary syntax[3].

Visualisation of code may either act as secondary syntax in order to enhance code comprehension for human viewers, or go further as primary syntax to enhance meaning for both humans and computers. The latter is of particular interest, as to some extent it requires making models of human perception the basis of computer language.

2.1. Morphology of sound, shape and symbols

TurTan is a geometric visual live coding language introduced by Gallardo et al. (2008), using the technology of the Reactable (Jordà et al., 2007). The functions of the language are manipulated as physical blocks that are placed on a tabletop interface, with nearest neighbours forming a sequence, and relative angle mapping to the function's parameter. The functions describe turtle graphics operations, and the resulting recursive forms are continuously updated on the table surface display.

TurTan inspired a system by Alex McLean and introduced here, with the working title of Acid Sketching. In Acid Sketching, a sound is specified simply by drawing a shape, where morphological measurements are mapped to parameters of an acid bass line synthesiser. The area of a shape is mapped to pitch, its regularity (perimeter length vs. area) mapped to envelope modulation, and relative angle of central axis mapped to resonance. Several such shapes are drawn in an arrangement, where a minimum spanning tree of their centroids is taken as a polyphonic sequence, where distance equals relative time. Feedback may be projected back on to the drawing surface, so shapes flash red as they are triggered. A static figure would not make this clearer, however illustrative video is available online at http://yaxu.org/acid-sketching/.

While Acid Sketching and TurTan are far from what is typically understood as live coding, both lead us to challenge understanding of the role of symbols, shape and geometry in computation. Investigating how such concrete forms of interaction could be married with the abstractions of general, Turing complete programming languages could be an interesting research topic itself.

Critically connected to live coding engagement with time-based media, is the time-based revelation of code itself. For electroacoustic music, Pierre Schaeffer's theories of sound timbre have been further dynamised into the time-variant sonic gestures of Denis Smalley's spectromorphology (Landy, 2007). For live coding, we might analogously dub 'codeo-morphology' as the changing shape of code over time. Examples include the accumulating code revisions referenced on the edge of ChucK language Audicle documents, or SuperCollider's 'History' class to document a live code performance. More visual representations of change over time would include accessible visualisations of programmer activity. Metrics might be displayed to characterise changes per second, from coarse keystroke counts to the depth of parse tree disruption; this brings us to self-evaluating performances, and coder re-coding of their very visualisations.

3. VISUAL EXPERIMENTS IN LIVE CODE

This section serves to introduce four novel visual/geometric live coding systems by Dave Griffiths, namely *Scheme Bricks*, *Betablocker*, *Al-Jazari* and *Daisy Chain*, along with some of the

systems which inspired them. All of these languages were constructed within Fluxus, a game engine designed for live coding performances and experiments and available under a free (GPL) license from http:// www.pawfal.org/fluxus/.

3.1. Execution flow and operational events

Computation is a metaphorical movement, where algorithmic processes operate on data (which can include the algorithms themselves) in memory in the discrete time steps of the CPU. Ways of visualising memory as it is changed have been developed for conventional debuggers, particularly in microcontroller applications where memory is small enough to be viewed in its entirety. More novel visualisations also exist, such as Tierra, an artificial life simulation where code evolves in a Darwinian competition which can only be appreciated when viewed as such, or Core War (Figure 3), a game where player/programmers write code which fight over memory address space.

Live coding has the unique opportunity to visualise the movement of an underlying process while it is being formed. This helps an audience appreciate a live coding performance in a more meaningful way – as it bridges the gap between an abstract description of a process (the code) and process itself (the generated pattern of movement through memory). Betablocker (Figure 4) is a raw visualisation of an imaginary 8-bit processor operating in 256 bytes of memory. This brightly coloured live coding environment is operated by writing assembly code with a gamepad. The processes are visualised while they operate on the memory addresses and trigger sound events. Processes are able to modify themselves and each other, resulting in highly dynamic relationships which are challenging to control.

A more traditional method of programming is employed in *Scheme Bricks* (Figure 6), a geometric interface for constructing Scheme programs.

```
vv<<
2
^ v<
v1<?>3v4
^^
>>?>?>5^
Vv
  v9<?>7v6
vv<
8
  .>> ^
  ^<
```

Figure 1: *A pseudo-random number generator written in the two-dimensional language Befunge*

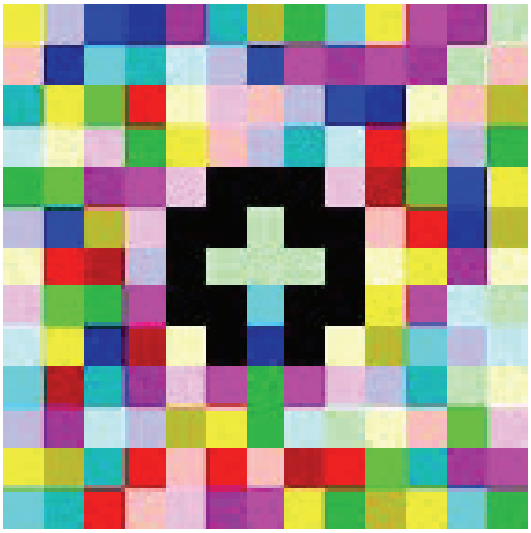

Figure 2: *Source code written in the Piet language with two dimensional, colour syntax. Prints out the text 'Hello, world!'. Image © Thomas Schoch 2006. Used under the Creative Commons BY-SA 2.5 licence*

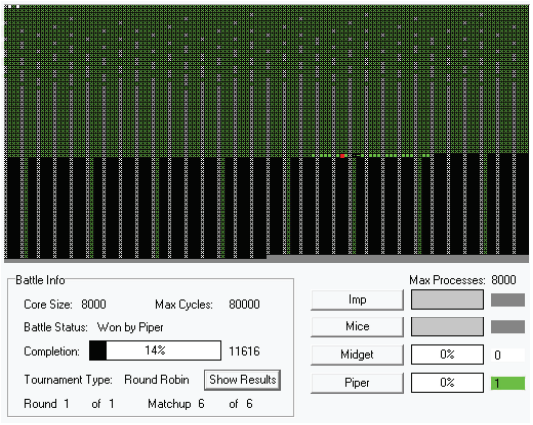

Figure 3: *Core war runtime display, showing visualisation of process memory shared between the players*

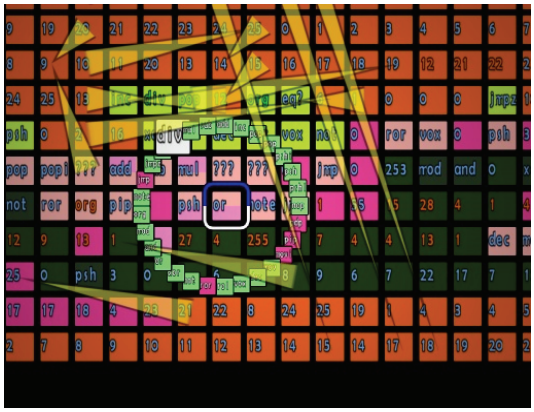

Figure 4: *A live edit in the Betablocker environment, selecting an instruction from a wheel of possibilities*

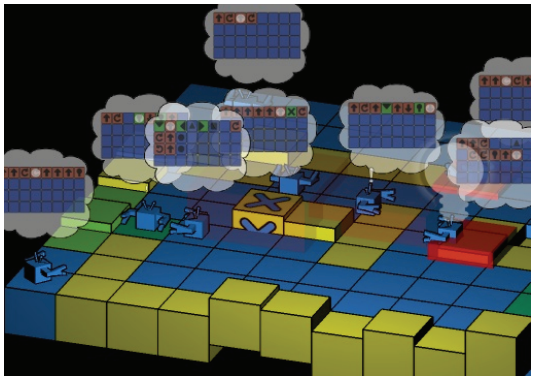

Figure 5: *The robots of Al-Jazari, each with a thought bubble containing a program, live coded with a gamepad*

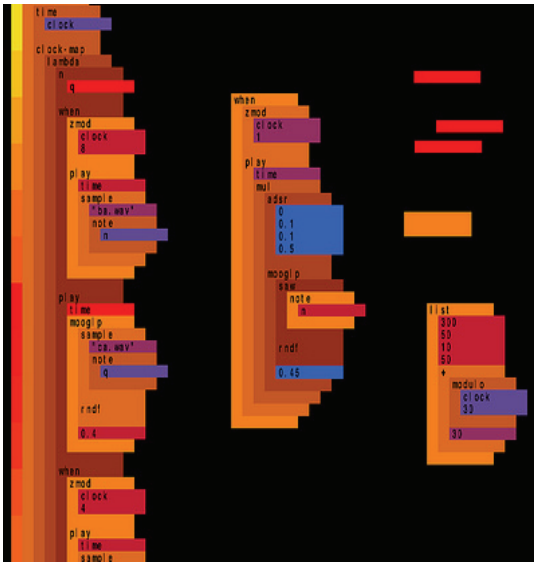

Figure 6: *Scheme Bricks, a lisp environment using colour instead of parenthesis, and flashes as a cue for control flow*

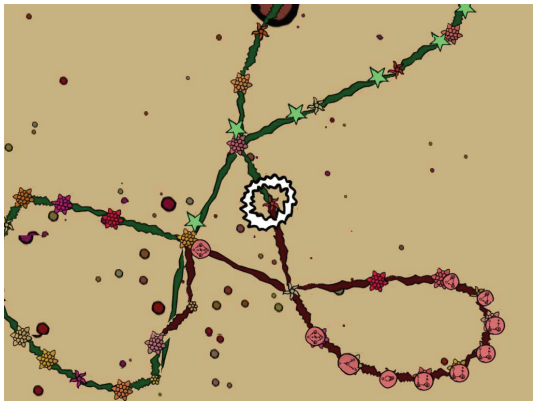

Figure 7: *A section of a Daisy Chain program*

Scheme Bricks takes advantage of the isomorphism of code and data in the Scheme programming language, and is inspired by the Scratch language designed for use by children (Resnick et al., 2009). Scheme Bricks allows you to drag, drop and plug together programs rather than typing. This has some potential side effects; in a performance situation, it is impossible to have a mismatched parenthesis error, as is common in other lisp-like languages. It is quicker to change the overall structure of the program as sections can be removed and reinserted easily by drag/drop actions. Unwanted sections are pulled out of the program and set aside rather than being deleted, and accumulate around the program as 'spare parts' which are often later 'recycled' by being pulled back into another section.

Scheme Bricks uses visual feedback to relate sound events to the code; the instruction which triggered a sound event flashes as the sound is played. This minimal approach to process visualisation makes the relationship between sound and code structure clearer than *Betablocker*'s more complete visualisation, and is useful for the performer to immediately locate the code generating a particular sound event.

Daisy Chain (Figure 7) is an attempt to embrace less rigid structures while maintaining enough of a computational basis to qualify as a live coding performance. It follows a processing system based on Petri nets (Petri, 1966), where executable instruction tokens move around a directed graph. Daisy Chain programs create and modify the graph topologies that they inhabit, producing sounds as a side effect of the computation. The look of the performance was designed to be as far from conventional programming as possible, hand animated flowers and drawn instruction symbols moving around graphs constrained by spring models.

The nodes of a Daisy Chain graph have a fixed lifetime, which was introduced in order to counter a common problem with live coding where the audience watching and performer concentrating on programming tend to perceive time differently. Daisy Chain prevents musical structures from persisting too long, keeping the performance moving forward at a rate the performer can control beforehand.

3.2. Computation in game worlds

Code has a long tradition of use in games as a gameplay mechanic, an early example being Core War developed in the mid 1980s and discussed above in section 3.1. More recent games such as Carnage Heart and Marionette Handler are mainstream games for the Playstation which employ programming environments using icons. These programs are used to control robots which battle it out in large virtual arenas. Popular game titles such as Little Big Planet allow the player to construct

machines as part of game worlds, complex enough to support Turing complete computation. Kodu, a research project at Microsoft goes even further, as an end-user games programming environment on the XBox.

Al-Jazari is a deliberate attempt to fuse games and live coding performances. It was designed to use a similar visual process to BetaBlocker, but this time mediated through the actions of robotic agents moving around a 3D world, triggering sounds as they do so (Figure 5). The use of visual agents following commands rather than abstract processes is intended to make the performance more immediately understandable for the audience. Al Jazari has been expanded as an art installation, audience participatory performance and recently as a Facebook game – with the aim to increase the accessibility of live coding to the point where anyone can become a live coder.

4. CONCLUSION

Visualisation is central to live coding. In this article, we have confronted how code is perceived by performers and audiences, and in what ways visual elements contribute to the primary syntax and semantics of a programming language meant for live coding. Consideration of visual elements of code have also become essential as live coding has formed the basis of virtual game worlds. We have introduced a number of novel systems, presented here as explorations of these themes. Visualisation of live code however remains under-investigated in terms of the psychology of programming; while Blackwell and Collins (2005) lead the way into HCI, evaluation protocols are yet to be adapted and applied to experience of live coded performances. This is, however, fertile ground for practice based research, and we anticipate the changing shapes of code over time, a codeomorphology at timescales from individual performances to lifetimes of artistic and technological development.

5. REFERENCES

Blackwell, A. and Collins, N. (2005). The programming language as a musical instrument. In *Proceedings of PPIG05*.University of Sussex.

Cascone, K. (2003). Grain, sequence, system (three levels of reception in the performance of laptop music). In Kleiner, M. S. and Szepanski, A.(eds) *Soundcultures*. Suhrkamp.

Collins, N., McLean, A., Rohrhuber, J., and Ward, A. (2003) Live coding in laptop performance. *Organised Sound*, 8(03) pp. 321–330.

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)*.

Gärdenfors, P(2000) *Conceptual Spaces: The Geometry of Thought.* The MIT Press.

Jordà, S., Geiger, G., Alonso, M. and Kaltenbrunner, M. (2007) Thereactable: Exploring the synergy between live music performance and tabletop tangible interfaces. In *Proc. Intl. Conf. Tangible and Embedded Interaction (TEI07)*.

Landy, L. (2007) *Understanding the Art of Sound Organization*. The MIT Press.

Petri, C. A. (1966) *Communication with automata. Technical report*, Applied Data Research Inc.

Puckette, M. (1988) Thepatcher. In *Proceedings of International Computer Music Conference*.

Resnick, M., Maloney, J., Hernández, A. M., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., and Kafai, Y. (2009) Scratch: programming for all. *Commun. ACM*, 52(11), pp. 60–67.

Wang, G. and Cook, P. R. (2004) On-the-fly programming: using code as an expressive musical instrument. In *NIME '04: Proceedings of the 2004 conference on New interfaces for musical expression*, pp. 138–143. Singapore, Singapore. National University of Singapore.

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., (eds) *read me — Software Art and Cultures*.

[1] A counter-example would be programming interfaces for the blind, which employ speech synthesis.

[2] The term 'secondary syntax' is problematic. Firstly, secondary syntax is only secondary relative to the computer interpreter, and not the human. Secondly, secondary syntax is not syntax in any clear sense; indeed spatial relationships are the basis of semantic meaning as understood in the field of cognitive linguistics. However as secondary syntax is the standard term used in the field of Human Computer Interaction (HCI) we persist with using it here.

[3] In Max, left-right position alters execution order, although relying upon this is discouraged in favour of the 'trigger' object.