TEXTURE: VISUAL NOTATION FOR LIVE CODING OF PATTERN

Alex McLean, Geraint Wiggins

Centre for Cognition, Computation and Culture
Department of Computing
Goldsmiths, University of London
alex@slab.org, g.wiggins@gold.ac.uk

ABSTRACT

Live coding, the use of programming language in improvised performance, is the subject of growing research interest. However little light has so far been thrown on the visual (as opposed to the temporal) aspects of live coding practice. Live coders project their code when they perform in the name of openness, but in so doing create a troubling issue of audience code comprehension. Relatedly, the constraining pressures of live performance are leading live coders to rethink the visual design of their language interfaces, so they may rework programs at greater speed and with lower cognitive load, using representations that are closer to music compositional structure. These two issues meet at the boundary between human perception and language. We examine this boundary to find high-level understanding of issues in the design of live coding languages, which is then practically applied in the introduction of Texture, a visual programming language for improvising music.

1. INTRODUCTION: FROM TIME TO SPACE – DIRECTIONS IN LIVE CODING

The definition of *live coding* used here is “the activity of writing a computer program while it runs”, after Ward et al. [1]. Closely related terms are *interactive*, *on-the-fly* [2], *conversational* [3], *just-in-time* [4] and *with-time* [5] programming. Many of these terms are interchangeable, although there are differences of technique and emphasis, for example the phrase *live coding* is most often used in the context of improvised performance of music or video animation. The archetypal live coding performance involves programmers writing code on stage, with their screens projected for an audience, their code dynamically interpreted to generate music or video. The particular context of improvised computer music is adopted here, and although much of the following could be related to work in live video animation, sole focus on computer music is kept for brevity.

The primary research focus around live coding practice has been upon the integration of performance time with development time, for example in the live coding papers already cited. This is important work, as the progression of time during the evaluation of an algorithm has often been purposefully ignored in computer science [6]. This line of research is certainly not complete, but there are now several working approaches to improvising music through live code development. Some research emphasis has therefore moved from time to space, that is, to the consideration of visuospatial perception within the activity and spectacle of live coding performance.

In the following we take a psychological perspective, finding a view that integrates visuospatial perception with the rather linguistic activity of computer programming. From this we outline design considerations for live coding systems, finally grounding the discussion with the introduction of a working prototype of a visual programming language, designed for live coding of musical pattern.

2. OBSCURANTISM IS DANGEROUS. SHOW US YOUR SCREENS.

The present section title is taken from the manifesto drafted by the Temporary Organisation for the Promotion of Live Algorithm Programming [TOPLAP; 1], a group set up by live coders to discuss and promote live coding. This bold proclamation neatly encapsulates a problem at the heart of live coding; live coders wish to show their interfaces so that the audience can see the movement and structure behind their music, however in positioning themselves against the computer music tradition of hiding behind laptop screens [7], they are at risk of a charge of greater obscurantism. Most people do not know how to program computers, and many who do will not know the particular language in use by a live coder. So, by projecting screens, do audience members feel *included* by a gesture of openness, or *excluded* by a gibberish of code in an obscure language? Do live coding performances foster melding of thoughts between performer and audience, or do they cause audience members to feel stupid? Audiences have not yet been formally surveyed on this issue, but anecdotal experience suggests both reactions are possible. A non-programmer interviewee in a BBC news item (“Programming, meet music”, 28th August 2009) reported ignoring projected screens and just listening to the music, and more negative reactions have been rumoured. On the other hand, a popular live coding tale has it that after enjoying a live coding performance by Dave Griffiths in Brussels (FoAM studios, 17th December 2005), a non-programmer turned to their lover and was overheard to exclaim “Now I understand! Now I understand why you spend so much time programming."

Partly in reaction to the issue of inclusion, a new direction of research into *visual programming* has emerged from live coding practice. To avoid confusion, note that the use of *visual* here is not the same as the prefix used in industry, such as in "Visual Basic", to describe language environments based on GUI forms and event-driven programming. Instead, *visual programming languages* are those making heavy use of visual elements in the code itself, for example where a program is notated as a graphical diagram. Visual programming language is itself a well established research field, with a great deal of promise but without wide industry take up. The intention of many visual language researchers has been to find ways of using visual notation that result in new, broadly superior general programming languages [8]. However this panacea has not been reached, and instead inherent, inescapable trade-offs have become apparent [9], which we will examine in §6. There are however two domains where visual programming has been highly successful, one being engineering, exemplified by the LabVIEW visual programming language. The other domain is, of course, computer music.

3. PATCHER LANGUAGES

It is time to stop avoiding the imposing glare from the elephant in the room. *Patcher* languages have been a dominant force in computer music since their introduction by Puckette [10]. Using a data flow model inspired by analog modular synthesis, users of Patcher languages such as Pure Data or Max/MSP are able to build patches using a visual notation of boxes and wires. Patches may be built and modified while they are active, a form of live coding that predates the contemporary live coding movement by well over a decade. The long-lived popularity of Patcher languages, the continued innovation within communities around them, and the artistic success of their use is undeniable.

While acknowledging the success of Patcher languages, we raise a point of controversy; in terms of syntax, they are not particularly visual. You can put an object wherever you like in PureData, or place them all on top of one another, it makes no difference. Those familiar with Max/MSP may counter this line of argument by pointing out that right-left ordering signifies evaluation order in Max. This falls on two counts; first, Max programmers are discouraged from relying on this, in favour of the *trigger* object. Second, having right-left execution order does not distinguish Max from any mainstream textual language. This lack of visual syntax in Patcher languages allows syntax graphs of hypercubes and up, *unconstrained* by visible dimensions. Furthermore, to say that Patcher languages are not themselves significantly textual is in blind denial of the large number of operators and keywords shown as editable text in a patch.

If the above seems like an attack on Patcher languages, then we immediately capitulate by pointing out that syntax is not everything. Well, to the interpreter syntax *is* everything, but to the programmer, it is only half the story. Every usable programming language has *secondary notation* [9], aspects such as comments, variable names and spatial arrangement with little or no syntactical significance, but which allow the programmer to write code that is readable and therefore maintainable. Many programming environments augment code with secondary notation not explicitly stored in a source file, such as colour syntax highlighting. Because Patcher languages have such remarkably free secondary notation, they allow us to lay programs out however we like, and to embrace this freedom in making beautiful patches that through shape and form relate structure to a human at a glance, in ways that linguistic syntax alone cannot do. While the language syntax is not visual, the notation as a whole is very much so.

4. CODE AND MENTAL IMAGERY

All programming languages can be considered in visual terms, we do after all normally use our eyes to read source code. Programming languages generally have context free-grammars, allowing recursive forms often encapsulated within parentheses, resulting in a kind of visual Euler diagram. We can also say that adjacency is a visual attribute of grammar; gestalt psychologists certainly have a lot to say about adjacency and perception [11]. These visual features generally exist to support linguistic reading of code, where our eyes saccade across the screen, recognising discrete symbols in parallel, chunked into words [12]. Crucially however, we are able to attend to both visuospatial and linguistic aspects of a scene simultaneously, and integrate them. Paivio [13] explains this through his theory of *dual coding* in humans, which considers mental imagery as supporting a kind of visuospatial thought separate from language. In dual coding, visuospatial cognition runs parallel to linguistic cognition, although the two systems support one another. For example, we gain information simultaneously from both spoken words and the prosodic manner in which they are articulated. We can straightforwardly relate this to programming languages; discrete symbols are expressed within linguistic grammar, supplemented by visuospatial arrangement expressing paralinguistic structure. The computer generally only attends to the first, but the human is able to attend to both.

Magnusson [14] describes a fundamental difference between acoustic and digital music instruments in the way we play them. He rightly points out that code does not vibrate, and so we cannot learn a computer music language with our bodies, in the same way as an acoustic instrument. However, programmers still have bodies which shape their thoughts, and in turn, through secondary notation, shape their code. Programmers do not physically resonate with code, but cognitive resources grounded in perceptual acuities enable them to take advantage of visuospatial cognition in their work.

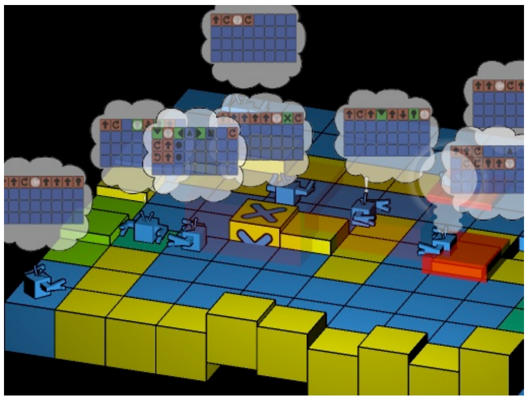


Figure 1. The robots of the Al-Jazari language by Dave Griffiths [16]. Each robot has a thought bubble containing a small program, edited through a game pad.

We have seen that visuospatial arrangement is of vital importance to the notation of Patcher languages, despite not being part of syntax. Our assertion follows that if shape, geometry and perceptual cues are so important to human understanding, then we should look for ways of taking these aspects out of secondary notation and make them part of primary syntax. Indeed, some languages, including recently developed music programming languages already have.

5. GEOMETRY AS SYNTAX

Artists often lead the way in technology, and programming language design is no exception. We are therefore able to highlight some examples of computer music languages which include geometrical measures of spatial arrangement in their primary syntax. Firstly, *Nodal* is a commercial environment for programming agent-based music [15]. Nodal has several interesting features, but is notable here for its spatial syntax, where distance symbolises elapsed time. As the graph is read by the interpreter, musical events at graph nodes are triggered, where the flow of execution is slowed by distance between nodes. Colour also has syntactic value, where paths are identified by one of a number of hues.

Al-Jazari is one of a series of playful languages created by Dave Griffiths, based on a computer game engine and controlled by a gamepad [16]. In Al-Jazari, cartoon depictions of robots are placed on a grid and given short programs for navigating it, in the form of sequences of movements including interactions with other robots. As with Nodal, space maps to time, but there is also a mechanism where robots take action based on the proximity and orientation of another robot. In programming Al-Jazari you are therefore put in the position of viewing a two dimensional space from the point of view of an agent's flow of execution. Indeed it is possible to make this literally so, as you may switch from the 'crows nest' view shown in Figure 1 to the 'first-person' view of a robot.

Our final example is the ReacTable [17], a celebrated 'tangible' interface aimed towards live music. Its creators do not describe the ReacTable as a programming language, and claim its tangible interface overcomes inherent problems in visual programming languages such as Pure Data. But truly, the ReacTable is itself a visual programming language, if an extraordinary one. It has a visual syntax, where physical blocks placed on the ReacTable are identified as symbols, and connected according to a nearest neighbour algorithm. Not only that, but relative distance and orientation between connected symbols are parsed as values, and used as parameters to the functions represented by the symbols. Video is back-projected onto the ReacTable surface to give feedback to the musician, for example by visualising the sound signal between nodes. The ReacTable has also been repurposed for an experimental interface for making graphics [18], suggesting that the ReacTable is not as far from a general programming language as it may first appear.

6. COGNITIVE DIMENSIONS OF NOTATION

Before moving to our practical contribution, we backtrack slightly to find support for our discussion from the psychology of programming literature. The Cognitive Dimensions of Notation (CDN) framework is designed to aid discussion of programming language features [9]. Rather than a checklist of good design, it describes a set of features which may be desirable or not, depending on the context. These are known as *dimensions*, indicating that they are not absolutes but scales. As illustrative examples, one dimension is *viscosity*, how easy it is to modify a program, and another is *closeness of mapping*, how related the programming notation is to the program output. If we change a notation to increase *closeness of mapping*, then *viscosity* is likely to increase, a factor which is usually undesirable. For brevity the reader is referred to Blackwell and Green [9] for full description of the dimensions, although thanks to the descriptive names given to the dimensions this is not strictly necessary for cursory understanding of the following.

The CDN is particularly useful to the design of Domain Specific Languages (DSLs), allowing consideration of the particular demands of a task domain in terms of trade-offs between notational features. Blackwell and Collins [19] have already examined the live coding domain with respect to the CDN, using it to compare the ChucK language [2] with the commercial Ableton Live production software. ChucK, and by implication live coding, does not come off particularly well. It has low *visibility*, *closeness of mapping* and *role-expressiveness*, is *error-prone* and requires *hard mental operations* in part to deal with its high level of *abstraction*. It would seem that the *progressive evaluation* and representational *abstraction* offered by ChucK come at a cost. Nonetheless, these are costs that many are willing to overcome through rigorous practice regimes reminiscent of instrumental virtuosos [20]. They are willing to do so because abstraction, while taking the improviser away from the direct manipulation that instrumentalists enjoy, allows them to focus on the compositional structure behind the piece. Being able to improvise music by manipulating compositional structure in theoretically unbound ways is too attractive a prospect for many to ignore.

Established norms place the live coder in a stage area separate from their audience members[1], who depending on the situation, may listen and watch passively or interact enthusiastically, perhaps by dancing, shouting or screaming. We therefore have two groups to consider; the performers needing to work 'in the moment' without technical interruptions that may break creative flow [23], and the audience members needing to feel included in the event, while engaged in their own creative process of musical interpretation. There is a challenge then in reconsidering live coding interfaces, creating new languages positioned at a place within the CDN well suited for a broader base of musicians and audiences who may wish to engage with them. The question is not just how musicians can adapt to programming environments, but also the inverse; how may programming environments, often designed to meet the needs of business and military institutions, be rethought to meet the particular needs of artists? First, we should consider what those needs might be.

An interesting cognitive dimension with respect to live coding is *error-proneness.* There are different flavours of error, some of which are much celebrated in electronic music, for example the *glitch* genre grew from an interest in mistakes and broken equipment [24]. In improvisation, an unanticipated outcome can provide a creative spark that leads a performance in a new direction. We would classify such desirable events as semantic aberrations, in contrast with syntactical errors which lead to crashes and hasty bug-fixing.

Turkle and Papert [25] draw a distinction between the programming style of *planners* and of *bricoleurs.* Bricolage programming is described in terms of the creative feedback loop adopted by many artists, who rather than having a separate design and implementation phase, instead design while implementing, deciding what to do next in response to every action. While Turkle and Papert [25] lack strong scientific grounding for many of their assertions [26], the idea of bricolage programming has relevance within a wide discourse. This feedback loop can be seen for example in the theory of creative motion related by the celebrated painter Paul Klee [27], and can be more generally related to theory of reflective practice in professional studies [28]. We follow our earlier work [6] in making a practical connection between bricolage programming and the arts. Bricolage is particularly relevant to the present theme of live coding, where 'blank slate' improvisations are the norm, with risk embraced and preplanning eschewed. The aim is to design a program 'in the moment' where it is implemented and executed for the (presumed) enjoyment of an audience.

[1]Performance norms are of course extensively challenged both inside [21] and outside [22] live coding practice.

In terms of the CDN, bricolage programming requires high *visibility* of components, in particular favouring shorter programs that fit on a single screen, and avoiding unnecessary *abstraction.* As noted above, abstraction sets live coding apart from other approaches to improvisation in computer music. However this is not an unnecessary layer of abstraction, but rather an important one that brings the programmer closer to their work. Programming after all is an activity that takes place somewhere between electronic transistors and lambda calculus – the bricolage programmer needs to find the right level of abstraction for their problem domain. Accordingly a computer musician may find having to deal with individual notes a distraction, and that a layer of abstraction above them provides the creative surface where they can feel closest to their composition.

7. INTRODUCING TEXTURE

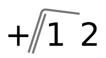
We now introduce Texture, a visual programming language for the live coding of pattern, designed predominantly for beat-driven techno performances. The name *Texture* is intended to accentuate the role of text in programming, as a structure woven into a two dimensional surface. An important design aim for Texture is to create a programming notation suitable for short, improvised scripts, allowing fast manipulation by an improviser, where lay audience members may appreciate more of the structure behind the code, made explicit through cues that are both visual and syntactical. Here we describe Texture as an early prototype system, describing the thinking behind it and issues raised through its development and early use.

The Texture environment and parser is implemented in C, using the free/open source Clutter graphics library (`http://clutter-project.org/`) for its user interface. It compiles into Haskell code, dynamically passed straight to the Haskell interpreter whenever the code is modified. Haskell then takes care of the task of evaluating the code and scheduling sound events accordingly, using the *Tidal* library created by the first author. Details of how the patterns are represented and transformed are given by McLean and Wiggins [29] and will not be repeated here. Much of Haskell's type system is re-implemented in Texture, the present contribution is not to provide a whole new language, but a complementary visual syntax.

The following is illustrated with real examples from the Texture user interface. Texture makes use of colour syntax highlighting, which for the purposes of printed proceedings has been switched off. We will however conclude with a colour example for the benefit of those reading via electronic means.

7.1. Geometric relationships

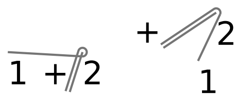
A program written in Texture is composed of strongly typed values and higher order functions. For example the function $+$ takes two numbers as arguments (either integers or reals, but not a mixture without explicit conversion), and returns their sum. Here is how `1 +2` looks in Texture:



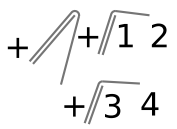
Two grey lines emerge from the bottom right hand side of the function $+$, both travelling to the upper left hand side of its first argument `1`, and then one travelling on to the second argument `2`. The significance of this will become clear later, but for now we note that the number of lines gives the function's arity[2].

The programmer types in functions and values, but does not manually add the lines connecting them, as they would with a Patcher language. The lines are instead drawn automatically by the language environment, inferred according to a simple rule: the closest two symbols connect, followed by the next two closest, and so on. Functions and values may be moved freely with the mouse, where a move may change the topology of the graph, which updates automatically. There is one important caveat here – symbols only connect if they are *type-compatible*, as we explain later.

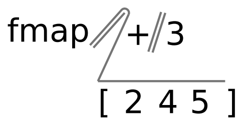
Texture has prefix notation (also known as Polish notation) where the function comes first, followed by its arguments, although the symbols can be placed and moved anywhere on the screen. For example `2 +1` may be expressed as either of the following:



Again, the symbols connect automatically, closest first, where 'closest' is defined as Euclidean distance in two dimensional space. If a symbol is moved, the whole program is re-interpreted, with connections re-routed accordingly. The functions may be composed together as you might expect. Here is `(1 +2) +(3 +4)` in Texture:



Texture is geared towards terse, higher order programming such as the following:



The $+$ function is only given a single value, despite having arity of two. The result is a function that (in this case) adds three to a given number. This function is an argument passed to *fmap*, which applies the function to the members of its second argument, a list. In other words, the function $+3$ is *mapped over* the members in the list `[2, 4, 5]`, which would result in `[5, 7, 8]`. A list is made simply by placing square brackets around values of the same type.

The use of $+3$ as a function deserves more explanation for those unfamiliar with this technique. This is made possible by functions being automatically *curried*, a feature taken from Haskell. This means that functions can be *partially applied*. In this example we *fix* the first argument as 3, getting back a new function which takes a single argument. This is what the visual lines in Texture represent, the application of each argument creating a new function with arity reduced by one. We believe this visual representation of curried functions to be novel, although would be pleased to hear of prior art.

The strong typing in Texture (again, taken from Haskell) places great restrictions on which arguments may be applied to which functions. This "bondage and discipline" works out well for Texture, as it limits the number of possible connections, making it easy for the programmer to predict what will connect where. They are aided further by colours used to identify types as shown in Figure 2 – only like colours connect. The `fmap` function is polymorphic, in that it can take any kind of function and apply it to any kind of list. But the strong typing means that those two kinds must be the same. For example, you cannot apply a string function to the elements of a list of integers, unless you also supply a conversion function. Through this mechanism, Texture enforces type correctness, avoiding all possibility of syntactically incorrect code.

7.2. User Interface

The Texture user interface is centred around typing, editing and moving words. In fact that is all you can do – there are no menus or key combinations. A new word is created by moving the cursor to an empty part of the screen using the mouse, and then typing. The word is completed by pressing the space bar, at which point the cursor moves a little to the right where the next word can be begun, mimicking a conventional text editor. A word is edited by being given focus with a click of the mouse, or moved by holding down the shift key while being dragged with the mouse. A whole function tree (the function and its connected arguments) is moved by holding down shift while dragging, although the arguments may connect differently in the new location according to the new context.

7.3. Musical Texture

Having seen much of the technical detail of Texture, we turn to its musical context. A video showing Texture in use is available at the website given at the end of this paper.

Texture is a prototype language that has not yet undergone full examination through HCI study, however preliminary observations have been conducted. In particular a small workshop for six participants was arranged with the Access Space free media lab in Sheffield, and led by the first author. The participants were self-selecting on a

[2] A function's *arity* is simply the number of arguments it requires

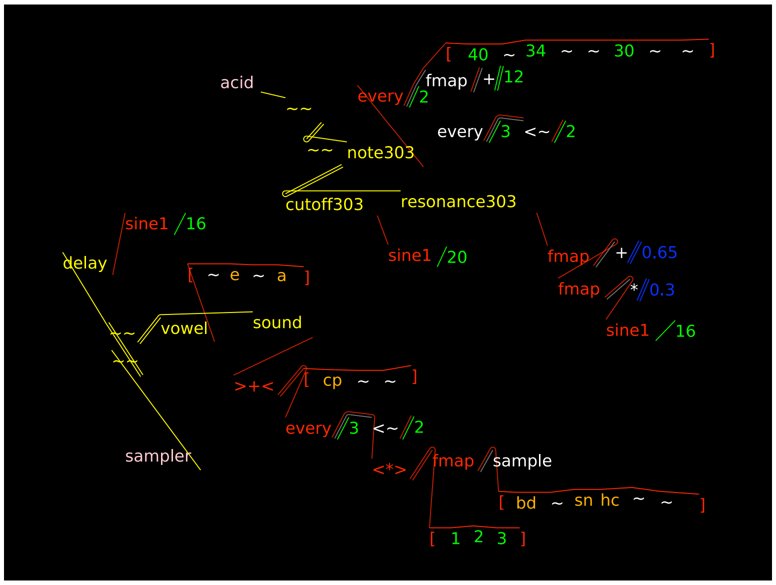


Figure 2. The Texture interface shown in colour, viewable via electronic proceedings. This program describes a layered drum loop with beat rotation, and a bass line with an octave shift, both with polyrhythmic variation of timbre/effects.

first-come-first-served basis, and all were male, between 23 and 42 years of age. Four lived locally to Sheffield, and two travelled from Liverpool. All were keen musicians, but only two had prior experience of programming. The workshop was free of charge, being part of a programme funded by the Arts Council, England. Participants were free to leave at any time with no penalty, but all stayed to the end.

The workshop was in the form of an hour long presentation surveying live coding practice and other influences of Texture, followed by a three hour hands-on workshop. The first half of the hands-on section introduced techniques on a projected display, which participants, while listening on headphones, copied and adapted in exploration of their use. The second half was more freeform, where each participant had their own set of stereo speakers at their computers. The participants were playing to a globally set tempo, with accurate time synchronisation.[3] This meant that they were able to respond to each other's patterns, improvising music together; because of the layout of the room, it was only really possible to clearly hear the music of immediate neighbours. Recorded video taken from this part of the workshop is available on the website given at the end of this paper.

The participants were the first people to use Texture besides the present author, and so there was some risk of unanticipated technical problems or task difficulty. However all showed enthusiasm, were keen to explore the language, and joined in with playing together over speakers.

The participants were surveyed for opinions through an anonymous on-line questionnaire. This was done in two parts, immediately before and then immediately after the workshop. They were asked to rate their agreement with three statements both before and after the workshop, on a scale from Disagree Strongly (1) to Agree Strongly (5). Although there is little statistical power for such a small group, feedback from these individuals was encouraging for a system at such an early stage of development. Agreement with "I am interested in live coding" fell slightly from 3.8 to 3.7. Agreement with "I am a live coder" rose from a mean of 1.5 to 3.1. Agreement with "I would like to be a live coder" was static at a mean of 3.5. A final statement given only at the end "I would like to use Text[ure] again" was met with mean agreement of 3.7.

Participants were also given freeform questions asking what they liked and disliked about Texture, how much they felt they understood the connection between the visual representation and the sound, and soliciting suggestions for improvements. Dislikes and suggestions focussed on technical interface issues such as the lack of 'undo', and three found the automatic linking difficult to work with. On the other hand, three participants reported liking how quick and easy it was to make changes.

7.4. Cognitive Dimensions of Texture

Texture is designed for the improvisation of musical pattern, as a visual programming interface to the Tidal pattern library [29]. The result is a more tightly constrained system than many programming languages for music, which include extensive facilities for low level sound synthesis. While the ability to compose right from the micro-level of the sound signal offers great possibilities, it comes with trade-offs, in particular along the *hard mental operations* and *diffuseness* (verbosity) CDNs.

The *visibility* of Texture is excellent, where a com-

[3]Accurate time synchronisation was made possible by the netclock protocol.

plex rhythm can be notated on a single screen. We find that Texture also has high *closeness of mapping*, as the visual representation of trees within trees corresponds well with the hierarchical structure of the pattern that is being composed. This echoes the tree structures common in music analysis, and indeed we would expect significant correspondence between the Texture structure and the listener's perception of it. The extent to which an untrained listener may relate the structure they hear with the Texture program they see is an empirical question, but we suspect that further development is needed to support this.

Creative use of Texture is aided not only by high *visibility* but also aspects of *provisionality*. A programmer may work on a section of code and drag it into the main body of the program when it is ready. They may also drag part of the code out of the main body and reuse it elsewhere later. The code must always be syntactically correct, but unless it connects to a function representing OSC messages sent to a synthesiser, it will have no effect.

The *error-proneness* of Texture is well positioned. It is impossible to make syntax errors in Texture, and while the automatic connection can at times have unexpected results, the result is at times musically interesting, but otherwise straightforward to reverse.

8. FUTURE DIRECTIONS

Texture is a working prototype, in that it is fully functional as a live music interface, but is a proof of concept of an approach to programming that brings many further ideas to mind.

In terms of visual arrangement, Texture treats words as square objects, but perhaps the individual marks of the symbols could be brought into the visual notation, through experiments in typography. For example, a cursive font could be used where the trajectory of the final stroke in a word is continued with a spline curve to flow into the leading stroke of the word it connects to. This suggestion may turn the stomach of hardened programmers, although Texture is already unusual in using a proportional font, complete with ligatures.

Currently there is no provision in Texture for making named abstractions, so a piece of code can only be used once in a program. Visual syntax for single assignment could symbolise a section of code with a shape derived from the arrangement of its components. That shape would become an ideographic symbol for the code, and then be duplicated and reused elsewhere in the program using the mouse.

Texture is inspired by the ReacTable, but does not feature any of the ReacTable's tangible interaction. Such tabletop interfaces offer a number of advantages over keyboard-and-mouse interfaces, in particular multitouch, allowing movement of more than one component at once. Multi-touch tablet computers share this advantage while avoiding some of the tradeoffs of tangible interfaces. Much of the ReacTable technology is available as an open research platform, and could be highly useful in this area of experimentation.

Currently the only output of Texture is music rendered as sound, with no visual feedback. There is great scope for experimentation in visualising the patterns in the code, making it easier for live coders and audience members to connect musical events and transformations with particular sections of the code. One approach would be the visualisation of pattern flowing between nodes, again inspired by the ReacTable, however as Texture is based upon pure functions rather than dataflow graphs, it presents a rather different design challenge. As Texture allows any Haskell program to be written, it could be applied to other domains, such as digital signal processing and visual animation. This would again place different challenges on the visualisation of results.

Most generally, and perhaps most importantly, we look towards proper analysis of lay audience code comprehension, grounding further development with better understanding of what the design challenges are.

9. CONCLUSION

We have considered programming languages as rich notations with both visual and linguistic aspects. Many computer musicians write words to describe their music, for computers to translate to sound. Computer musicians have become comfortable with this rather odd process in private, but perhaps found it difficult to explain to their parents. Simply by projecting their interfaces, live coders have brought this oddity out in public, and must deal with the consequences of bringing many issues underlying computer music to the surface.

Live coding performance should be understood in terms of the dual activity of language and spatial perception. We have seen how humans have the capacity to integrate both simultaneously, showing that the act of live coding, and perhaps the audience reception of it, can be realised and felt simultaneously as both musical language and musical movement.

We have provided psychological and analytical background to the design of live coding languages, as signposts for future work. We hope the introduction of Texture demonstrates the exciting ground waiting for future languages to explore.

Full source code for Texture and supporting libraries is available under the GNU Public License version 3, from `http://yaxu.org/category/texture/`.

10. REFERENCES

[1] A. Ward, J. Rohrhuber, F. Olofsson, A. McLean, D. Griffiths, N. Collins, and A. Alexander, "Live algorithm programming and a temporary organisation for its promotion," in *read_me — Software Art and Cultures*, O. Goriunova and A. Shulgin, Eds., 2004.

[2] G. Wang and P. R. Cook, "On-the-fly programming: using code as an expressive musical instrument," in

Proceedings of the 2004 conference on New interfaces for musical expression. National University of Singapore, 2004, pp. 138–143.

[3] I. Kupka and N. Wilsing, *Conversational Languages.* John Wiley and Sons, 1980.

[4] J. Rohrhuber, A. de Campo, and R. Wieser, "Algorithms today: Notes on language design for just in time programming," in *Proceedings of the 2005 International Computer Music Conference*, 2005.

[5] A. Sorensen and H. Gardner, "Programming with time: cyber-physical programming with impromptu," in *Proceedings of ACM OOPLSA*, 2010, pp. 822–834.

[6] A. McLean and G. Wiggins, "Bricolage programming in the creative arts," in *22nd Psychology of Programming Interest Group*, 2010.

[7] N. Collins, "Generative music and laptop performance," *Contemporary Music Review*, vol. 22, 2003.

[8] A. F. Blackwell, "Metacognitive theories of visual programming: what do we think we are doing?" in *Proceedings of IEEE Symposium on Visual Languages*, 2006, pp. 240–246.

[9] A. Blackwell and T. Green, *Notational Systems – the Cognitive Dimensions of Notations framework.* Morgan Kaufmann, 2002, pp. 103–134.

[10] M. Puckette, "The patcher," in *Proceedings of International Computer Music Conference*, 1988.

[11] W. Kohler, *Gestalt Psychology.* Camelot Press, 1930.

[12] K. Rayner and A. Pollatsek, *Word Perception.* Routledge, Nov. 1994, ch. 3.

[13] A. Paivio, *Mental Representations: A Dual Coding Approach (Oxford Psychology Series).* Oxford University Press, USA, Sep. 1990.

[14] T. Magnusson, "Of epistemic tools: musical instruments as cognitive extensions," *Organised Sound*, vol. 14, no. 2, pp. 168–176, 2009.

[15] P. Mcilwain, J. Mccormack, A. Dorin, and A. Lane, "Composing with nodal networks," in *Proceedings of the Australasian Computer Music Conference 2005*, T. Opie and A. Brown, Eds., 2005, pp. 96–101.

[16] A. McLean, D. Griffiths, N. Collins, and G. Wiggins, "Visualisation of live code," in *Proceedings of Electronic Visualisation and the Arts London 2010*, 2010.

[17] S. Jordà, G. Geiger, M. Alonso, and M. Kaltenbrunner, "The reacTable: Exploring the synergy between live music performance and tabletop tangible interfaces," in *Proc. Intl. Conf. Tangible and Embedded Interaction (TEI07)*, 2007.

[18] D. Gallardo, C. F. Julià, and S. Jordà, "TurTan: a tangible programming language for creative exploration," in *Third annual IEEE international workshop on horizontal human-computer systems (TABLETOP)*, 2008.

[19] A. Blackwell and N. Collins, "The programming language as a musical instrument," in *Proceedings of PPIG05.* University of Sussex, 2005.

[20] N. Collins, "Live coding practice," in *Proceedings of New Interfaces for Musical Expression 2007*, 2007.

[21] J. Rohrhuber, A. de Campo, R. Wieser, J.-K. van Kampen, E. Ho, and H. Hölzl, "Purloined letters and distributed persons," in *Music in the Global Village Conference*, 2007.

[22] C. Small, *Musicking: The Meanings of Performing and Listening (Music Culture)*, 1st ed. Wesleyan, Jun. 1998.

[23] M. Csikszentmihalyi, *Flow: the psychology of optimal experience.* HarperCollins, 2008.

[24] K. Cascone, "The aesthetics of failure: "Post-Digital" tendencies in contemporary computer music," *Computer Music Journal*, vol. 24, no. 4, pp. 12–18, 2000.

[25] S. Turkle and S. Papert, "Epistemological pluralism and the revaluation of the concrete," *Journal of Mathematical Behavior*, vol. 11, no. 1, pp. 3–33, Mar. 1992.

[26] A. Blackwell, "Gender in domestic programming: From bricolage to séances d'essayage," in *CHI Workshop on End User Software Engineering*, 2006.

[27] P. Klee, *Pedagogical sketchbook.* Faber and Faber, 1953.

[28] D. A. Schon, *The Reflective Practitioner: How Professionals Think In Action*, 1st ed. Basic Books, Sep. 1984.

[29] A. McLean and G. Wiggins, "Tidal - pattern language for the live coding of music," in *Proceedings of the 7th Sound and Music Computing conference*, 2010.