Bricolage Programming in the Creative Arts

Alex McLean and Geraint Wiggins

Centre for Cognition, Computation and Culture
Department of Computing
Goldsmiths, University of London

Abstract. In this paper we consider artists who create their work by writing algorithms, which when interpreted by a computer generates their plotted drawings, synthesised music, animated digital video, or whatever target medium they have chosen. We examine the demands that such artists place upon their environments, the relationships between concepts and algorithms, and of cognition and computation. We begin by considering an artist's creative process, and situating it within the bricolage style of programming. An embodied view of bricolage programming is related, underpinned by theories of cognitive metaphor and computational creativity, and finally with consideration of the bricolage programmer's relation to time.

1 Introduction

Over the last decade, computer programming has enjoyed a major resurgence as a medium for the arts. A wealth of new programming environments for the arts, such as Processing, SuperCollider, ChucK, VVVV and OpenFrameworks have joined more established environments such as PureData and Max which have themselves gained enthusiastic adoption outside their traditional academic base. These environments offer varied approaches to supporting artistic use, including alternative programming languages, interfaces and workflow.

The purpose of the present discussion is to examine psychological issues which the resurgence of artistic programming has brought to the fore. What is the relationship between an artist, their creative process, their program, and their artistic works? We will look for answers from perspectives of psychology, cognitive linguistics, computer science and computational creativity, but first from the perspective of the artist.

2 Creative Processes

The painter Paul Klee [1953, p. 33] describes a creative process as a feedback loop: "Already at the very beginning of the productive act, shortly after the initial motion to create, occurs the first counter motion, the initial movement of receptivity. This means: the creator controls whether what he has produced so far is good. The work as human action (genesis) is productive as well as receptive. It is **continuity**." This is creativity without planning, a feedback loop of making a mark on canvas, perceiving the effect, and reacting with a further mark. Being engaged in a tight creative feedback loop places the artist close to their work, guiding an idea to unforeseeable conclusion through a flow of creative perception and action. Klee writes as a painter, working directly with his medium. Programmer-artists instead work using computer language as a textual representation of their medium, and it might seem that this extra level of abstraction could hinder creative feedback. We will see however that this is not necessarily the case, beginning with the account of Turkle and Papert [1992], describing a *bricolage* approach to programming by analogy with painting:

> The bricoleur resembles the painter who stands back between brushstrokes, looks at the canvas, and only after this contemplation, decides what to do next. Bricoleurs use a mastery of associations and interactions. For planners, mistakes are missteps; bricoleurs use a navigation of midcourse corrections. For planners, a program is an instrument for premeditated control; bricoleurs have goals but set out to realize them in the spirit of

a collaborative venture with the machine. For planners, getting a program to work is like "saying one's piece"; for bricoleurs, it is more like a conversation than a monologue. [Turkle and Papert, 1990, p. 136]

Although Turkle and Papert address gender issues in education, this quote should not be misread as dividing all programmers into two types; while associating bricolage with feminine and planning with male traits, they are careful to note that these are extremes of a behavioural continuum. Indeed, programming style is clearly task specific: for example a project requiring a large team needs more planning than a short script written by the end user.

Bricolage programming seems particularly applicable to artistic tasks, such as writing software to generate music, video animation or still images. Imagine a visual artist, programming their work using Processing. They may begin with an urge to draw superimposed curved lines, become interested in a tree-like structure they perceive in the output of their first implementation, and change their program to explore this new theme further. The addition of the algorithmic step would appear to affect the creative process as a whole, and we seek to understand how in the following.

2.1 Creative Process of Bricolage

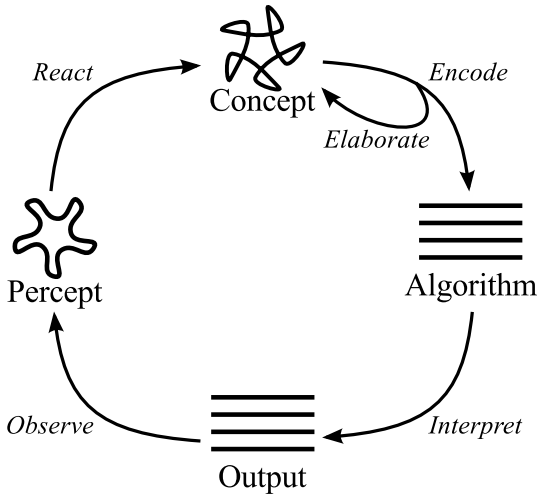

Fig. 1. The process of action and reaction in bricolage programming

Figure 1 shows bricolage programming as a creative feedback loop encompassing the written algorithm, its interpretation, and the programmer's perception and reaction to its output or behaviour. The addition of the algorithmic component in the creative feedback loop makes an additional inner loop explicit between the programmer and their text. At the beginning, the programmer may have a half-formed concept, which only reaches internal consistency through the process of being expressed as an algorithm. The inner loop is where the programmer elaborates upon their imagination of what might be, and the outer where this trajectory is grounded in the pragmatics of what they have actually made. Through this process both algorithm and concept are developed, until the programmer feels they accord with one another or otherwise judges the creative process to be finished.

Representations in the computer and the brain are evidently distinct from one another. Computer output evokes perception, but that percept will both exclude features that are explicit in the output and include features that are not, due to a host of effects including attention, knowledge and illusion. Equally, a human concept is distinct from a computer algorithm. Perhaps

a program written in a declarative rather than imperative style is somewhat closer to a concept, being not an algorithm for how to carry out a task, but rather a description of what is to be done. But still, there is a clear line to be drawn between a string of discrete symbols in code and the morass of symbolic, spatial and relational representations we assume underlies cognition.

There is however something curious about how the programmer's creative process spawns a second, computational one. The computational process is lacking in the cognitive abilities of its author, but is nonetheless both faster and more accurate at certain tasks by several orders of magnitude. It would seem that the programmer uses the programming language and its interpreter as a cognitive resource, augmenting their own abilities in line with the extended mind hypothesis [Clark, 2008]. We will revisit this issue within a formal framework in §5, after first looking more broadly at how we relate programming to human experience, and related issues of representation.

3 Anthropomorphism and Metaphor in Programming

Metaphor permeates our understanding of programming. Perhaps this is due to the abstract nature of computer programs, requiring metaphorical constructs to allow us to ground programming language in everyday reasoning. Petre and Blackwell [1999] gave subjects programming tasks, and asked them to introspect upon their imagination while they worked. These self reports are rich and varied, including exploration of a landscape of solutions, dealing with interacting creatures, transforming a dance of symbols, hearing missing code as auditory buzzing, combinatorial graph operations, munching machines, dynamic mapping and conversation. While we cannot rely on these introspective reports as authoritative on the inner workings of the mind, the diversity of response hints at highly personalised creative processes, related to physical operations in visual or sonic environments. It would seem that a programmer uses metaphorical constructs defined largely by themselves and not by the computer languages they use. However mechanisms for sharing metaphor within a culture do exist. Blackwell [2006a] used corpus linguistic techniques on programming language documentation in order to investigate the conceptual systems of programmers, identifying a number of conceptual metaphors listed in Figure 2. Rather than finding metaphors supporting a mechanical, mathematical or logical approach as you might expect, components were instead described as actors with beliefs and intentions, being social entities acting as proxies for their developers.

The above research suggests that programmers understand the operation of their programs by metaphorical relation to their experience as a human. Indeed the feedback loop described in §2 is by nature anthropomorphic; by embedding the development of an algorithm in a human creative process, the algorithm itself becomes a human expression. Dijkstra [1988] strongly opposed such approaches: "I have now encountered programs wanting things, knowing things, expecting things, believing things, etc., and each time that gave rise to avoidable confusions. The analogy that underlies this personification is so shallow that it is not only misleading but also paralyzing." Dijkstra's claim is that by focusing on the operation of algorithms, the programmer submits to a combinatorial explosion of possibilities for how a program might run; not every case can be covered, and so bugs result. Dijkstra argues for a strict, declarative approach to computer science and programming in general, which he views as so radical that we should not associate it with our daily existence, or else limit its development and produce bad software.

The alternative view presented here is that metaphors necessarily structure our understanding of computation. This view is sympathetic to a common assumption in the field of cognitive linguistics, that our concepts are organised in relation to each other and to our bodies, through conceptual systems of metaphor. Software now permeates Western society, and is required to function reliably according to human perception of time and environment. Metaphors of software as human activity are therefore becoming ever more relevant.

Components are agents of action in a causal universe.
Programs operate in historical time.
Program state can be measured in quantitative terms.
Components are members of a society.
Components own and trade data.
Components are subject to legal constraints.
Method calls are speech acts.
Components have communicative intent.
A component has beliefs and intentions.
Components observe and seek information in the execution environment.
Components are subject to moral and aesthetic judgement.
Programs operate in a spatial world with containment and extent.
Execution is a journey in some landscape.
Program logic is a physical structure, with material properties and subject to decay.
Data is a substance that flows and is stored.
Technical relationships are violent encounters.
Programs can author texts.
Programs can construct displays.
Data is a genetic, metabolizing lifeform with body parts.
Software tasks and behaviour are delegated by automaticity.
Software exists in a cultural/historical context.
Software components are social proxies for their authors.

Fig. 2. Conceptual metaphors derived from analysis of Java library documentation by Blackwell [2006a]. Program components are described metaphorically as actors with beliefs and intentions, rather than mechanical imperative or mathematical declarative models.

4 Symbols and Space

We now turn our attention to how the components of the bricolage programming process shown in Figure 1 are represented, in order to ground understanding of how they may interrelate. Building upon the anthropomorphic view taken above, we propose that in bricolage programming, the human cognitive representation of programs centres around perception. Perception is a low dimensional representation of sensory input, giving us a somewhat coherent, spatial view of our environment. By spatial we do not just mean in terms of physical objects, but also in terms of features in the spaces of all possible tastes, sounds, tactile textures and so on. This scene is built through a process of dimensional reduction from tens of thousands of chemo-, photo-, mechano- and thermoreceptor signals. Algorithms on the other hand are represented in discrete symbolic sequences, as is their output, which must go through some form of digital-to-analogue conversion before being presented to our sensory apparatus, for example as light from a monitor screen or sound pressure waves from speakers, a process we call observation. Recall the programmer from §2, who saw something not represented in the algorithm or even in its output, but only in their own perception of the output; observation may itself be a creative act.

The component from Figure 1 not yet mentioned in this section is that of programmers' concepts. A concept is 'a mental representation of a class of things' [Murphy, 2002, p.5]. Figure 1 shows concepts mediating between spatial perception and symbolic algorithms, leading us to ask; are concepts represented more like spatial geometry, like percepts, or symbolic language, like algorithms? Our focus on metaphor leads us to take the former view, that conceptual representation is grounded in perception and the body. This view is taken from Conceptual Metaphor Theory (CMT) introduced by Lakoff and Johnson [1980], which proposes that concepts are primarily structured by metaphorical relations, the majority of which are orientational, understood relative to the human body in space or time. In other words, the conceptual system is grounded in the perceptual system. Gärdenfors [2000] builds upon this by further proposing that the semantic meanings of concepts and the metaphorical relationships between them are geometrical properties and relationships. Concepts themselves are represented by geometric regions of low

dimensional spaces defined by quality dimensions, either mapped directly from, or structured by metaphorical relation to perceptual qualities. For example "red" and "blue" are regions in perceptual colour space, and the metaphoric semantics of concepts within the spaces of mood, temperature and importance may be defined relative to geometric relationships of such colours.

Gärdenforsian conceptual spaces are compelling when applied to concepts related to bodily perception, emotion and movement, and Forth et al. [2008] report early success in computational representations of conceptual spaces of musical rhythm and timbre, through reference to research in music perception. However, it is difficult to imagine taking a similar approach to computer programs. What would the quality dimensions of a geometrical space containing all computer programs be? There is no place to begin to answer this question; computer programs are symbolic in nature, and cannot be coherently mapped to a geometrical space grounded in perception.

For clarity we turn once again to Gärdenfors [2000], who points out that spatial representation is not in opposition to symbolic representation; they are distinct but support one another. This is clear in computing, hardware exists in our world of continuous space, but thanks to reliable electronics, conjures up a symbolic world of discrete computation. Our minds are able to do the same, for example by computing calculations in our head, or encoding concepts into phonetic movements of the vocal tract or alphabetic symbols on the page. We can think of ourselves as spatial beings able to simulate a symbolic environment to conduct abstract thought and open channels of communication. On the other hand, a piece of computer software is a symbolic being able to simulate spatial environments, perhaps to create a game world or guide robotic movements, both of which may include some kind of model of human perception.

Computer language operates in the domain of abstraction and communication but in general does not at base include spatial semantics. In some cases computer languages are described as 'visual' even when spatial arrangement is purely secondary notation, ignored by the interpreter, such as in Patcher languages [Puckette, 1988]. In fact spatial layout is a feature of secondary notation in mainstream 'textual' languages too, through use of whitespace with no syntactical meaning. That programmers need to use spatial layout as a crutch while composing symbolic sequences is telling; to the interpreter, a block may be a subsequence between braces, but to an experienced programmer it is a perceptual gestalt grouped by indentation. From this we can understand computation as separate from spatial reasoning, but supported by it, with secondary notation helping bridge the divide.

An important aspect of CMT is that a conceptual system of semantic meaning exists within an individual, not in the world. Through language, metaphors become established in a culture and shared by its participants, but this is an effect of individual conceptual systems interacting, and not individuals inferring and adopting external truths of the world (or of possible worlds). This would account for the varied range of metaphor in programming discussed in §3, as well as the general failure of attempts at designing metaphor into computer interfaces [Blackwell, 2006b]. Each programmer has a different set of worldly interests and experiences, and so establishes different metaphorical systems to support their programming activities.

5 Components of creativity

We now have sufficient grounds to fully characterise how the creative process operates in our case study of bricolage programming. For this we employ the Creative Systems Framework (CSF), a high-level formalisation of creativity introduced by Wiggins [2006a,b] and based upon the work of Boden [2003]. Creativity is characterised as a search in a space of concepts. Three sets of rules are employed in this search; $\mathcal{R}$ defining the *search space* itself, $\mathcal{T}$ defining *traversal* of the space and $\mathcal{E}$ defining *evaluation* of concepts found in the space. However, the CSF describes much more than a reactive process of traversal and evaluation. Creativity also requires introspection, self-modification and for boundaries to be broken. In other words, the rulesets $\mathcal{R}$, $\mathcal{T}$ and $\mathcal{E}$ are examined and challenged by the creative agent following them.

Using the terms of Gärdenfors [2000], $\mathcal{R}$ is a concept defining a space of concept instances.[1] For example in a creative search for music within a genre, the genre would be the concept and a piece of music conforming to a genre would be a instance of that concept. Crucially, $\mathcal{R}$ is not a closed space, but rather defined as a subspace of the universe of all possible concepts. This means that a creative agent may creatively push beyond the boundaries of the search as we will see.

We are now in a position to clarify the bricolage programming process introduced in §2.1 within the CSF. As shown in Figure 3, the ruleset $\mathcal{R}$ defines the programmer's concept, being their current artistic focus structured by learnt techniques and conventions, the traversal strategy $\mathcal{T}$ is the process of encoding and interpreting the algorithm, and the evaluation function $\mathcal{E}$ is the perceptual process of observation and reaction.

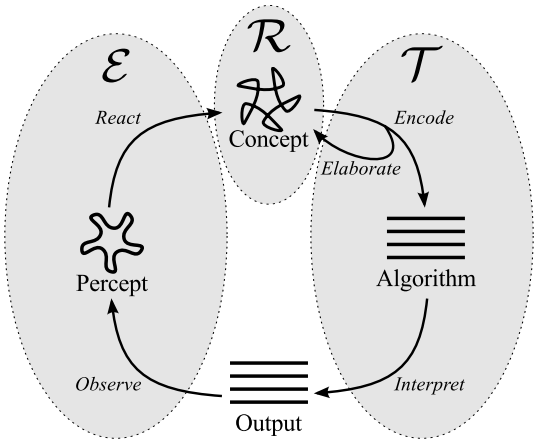

Fig. 3. The process of action and reaction in bricolage programming from Figure 1, annotated with the $\mathcal{R}$ conceptual space, $\mathcal{T}$ traversal strategy and $\mathcal{E}$ evaluation components of the Creative Systems Framework.

In §2, we alluded to the extended mind hypothesis [Clark, 2008], claiming that bricolage programming takes part of the human creative process outside of the mind and into the computer. The above makes clear what we claim is being externalised: part of the traversal strategy $\mathcal{T}$. The programmer's concept $\mathcal{R}$ motivates a development of the strategy $\mathcal{T}$ to be encoded in a program, but the programmer does not necessarily have the cognitive ability to fully evaluate the program. That task is taken on by the interpreter running on a computer system, meaning that $\mathcal{T}$ encompasses both encoding by the human and interpretation by the computer.

The traversal strategy $\mathcal{T}$ is structured by the techniques and conventions employed to convert concepts into operational algorithms. These may include *design patterns*, a standardised set of *ways of building* that have become established around imperative programming languages. Each design pattern identifies a *kind* of problem, and describes a *kind* of structure as a *kind* of solution.[2]

The creative process is constrained by $\mathcal{R}$, being the programmer's idea of what is a valid end result. This is shaped by the programmer's current artistic focus, being the perceptual qualities they are currently interested in, perhaps congruent with a cultural theme such as a musical genre or artistic movement. Transformational creativity can be triggered in the CSF when application of $\mathcal{T}$ results in a concept instance that exists outside the constraining bounds of $\mathcal{R}$, shown in Figure 4. If the instance is valued according to $\mathcal{E}$, then $\mathcal{R}$ is changed to include

[1] The terms used by Gärdenfors [2000] diverge from those used by Wiggins [2006a,b]. Wiggins uses the term *conceptual space* in the place of Gärdenfors' *concept*, and *concept* in the place of *concept instance*. The meaning is however the same, particularly when the recursive hierarchy of Wiggins' theory is taken into account.

[2] Interestingly, this structural heuristic approach to problem solving originated in the field of urban design [Alexander et al., 1977].

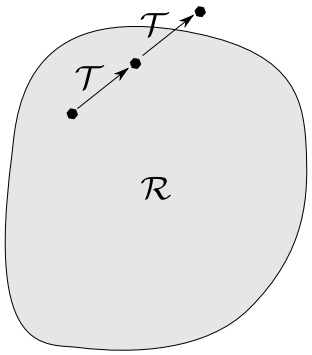

Fig. 4. Application of a traversal strategy $\mathcal{T}$ leading outside the concept $\mathcal{R}$, triggering transformational creativity.

it. If the instance is not valued, then $\mathcal{T}$ is changed to avoid that instance in the future. As a result of including external interpretation in $\mathcal{T}$, the programmer is likely to be less successful in writing software that meets their preconceptions, but as a result more successful in being surprised by the results. In other words, the artist's act of externalising part of $\mathcal{T}$ as a computer program makes the results less predictable, and transformational creativity more likely.

In artistic bricolage programming, then, we conclude that creativity is a process of imagining a concept $\mathcal{R}$, encoding an operational algorithm as part of $\mathcal{T}$ to explore within and beyond $\mathcal{R}$, and a perceptual process $\mathcal{E}$ to evaluate the output. Through this process both $\mathcal{R}$ and $\mathcal{T}$ are continually transformed in respect of one another, in creative feedback.

According to our embodied view, not only is perception crucial in evaluating output within bricolage programming, but also in structuring the space in which programs are conceptualised. Indeed if the embodied view of CMT holds in general, the same would apply to all creative endeavour. From this we find a message for the field of computational creativity: a prerequisite for an artificial creative agent is in acquiring computational models of perception sufficient to both evaluate its own works and structure its conceptual system. Only then will the agent have a basis for guiding changes to its own conceptual system and generative traversal strategy, able to modify itself to find artifacts that it was not programmed to find, and place value judgements on them. Such an agent would need to adapt to human culture in order to interact with shifting cultural norms, keeping its conceptual system and resultant creative process as coherent within that culture. For now however this is all wishful thinking, and we must be content with generative computer programs which extend human creativity, but are not creative agents in their own right.

6 Programming in Time

> "She is not manipulating the machine by turning knobs or pressing buttons. She is writing messages to it by spelling out instructions letter by letter. Her painfully slow typing seems laborious to adults, but she carries on with an absorption that makes it clear that time has lost its meaning for her." Sherry Turkle [2005, p. 92], on Robin, aged 4, programming a computer.

Having investigated the representation and operation of bricolage programming we now examine how the creative process operates in time. Considering computer programs as operating in time at all, rather than as logic abstract from the world, is itself a form of the anthropomorphism examined in §3. However from the above quotation it seems that Robin stepped out of any notion of physical time, and into the algorithm she was composing, entering a timeless state. Speaking anecdotally, programmers report losing hours as they get 'in the flow' when writing software. Perhaps a programmer is thinking in algorithmic time, attending to control flow as it replays over and over in their imagination, and not to the world around them. Or perhaps they

are not attending to the passage of time at all, thinking entirely of declarative abstract logic, in a timeless state of building. In either case, it would seem that the human is entering time relationships of their software, rather than the opposite, anthropomorphic direction of software entering human time. However there are ways in which human and computational time may be united, which we will come to shortly.

Temporal relationships are generally not represented in source code. When a programmer needs to do so, for example as an experimental psychologist requiring accurate time measurements, or a musician needing accurate synchronisation between processes, they run into problems. With the wide proliferation of interacting embedded systems, this is becoming a broad concern [Lee, 2009]. In commodity systems time has been decentralised, abstracted away through layers of caching, where exact temporal dependencies and intervals between events are not deemed worthy of general interest. Programmers talk of 'processing cycles' as a valuable resource which their processes should conserve, but they generally no longer have programmatic access to the high frequency oscillations of the central processing units (now, frequently plural) in their computer. The allocation of time to processes is organised top-down by an overseeing scheduler, and programmers must work to achieve what timing guarantees are available. All is not lost however, realtime kernels are now available for commodity systems, allowing psychologists [Finney, 2001] and musicians (e.g. via `http://jackaudio.org/`) to get closer to physical time. Further, the representation of time semantics in programming is undergoing active research in a subfield of computer science known as *reactive programming* [Elliott, 2009], with applications emerging in music [McLean and Wiggins, 2010].

6.1 Interactive programming

Programmers who 'think' in algorithmic time, like Robin earlier, are well served by dynamically interpreted languages. These allow a programmer to examine an algorithm while it is interpreted, taking on live changes without restarts. This is known as *interactive programming*, and unites the time flow of a program with that of its development. Interactive programming makes a dynamic creative process of test-while-implement possible, rather than the conventional implement-compile-test cycle, so that arrows shown in Figures 1 and 3 show concurrent influences between components rather than time-ordered steps.

Interactive programming not only provides a more efficient creative feedback loop, but also allows a programmer to connect software development with time based art. Since 2003 an active group of practitioners and researchers have been developing new approaches to making computer music and video animation, collectively known as *Live coding* [Blackwell and Collins, 2005, Ward et al., 2004, Collins et al., 2003, Rohrhuber et al., 2005]. The archetypal live coding performance involves programmers writing code on stage, with their screens projected for an audience, while the code is dynamically interpreted to generate music or video. Here the process of development is the performance, with the work generated not by a finished program, but its journey of development from an empty text editor to complex algorithm, generating continuously changing musical or visual form along the way. This is bricolage programming perhaps taken to a logical and artistic conclusion.

7 Conclusion

What we have seen provides strong motivation for programming which address the concerns of artists. These include concerns of workflow, where any time elapsed between source code edit and interpreted output is slows the creative process. Concerns of interfaces are also important, where in certain situations greater emphasis is placed on presentation of short scripts in their entirety as per bricolage programming, rather than hierarchical views of larger codebases. Perhaps most importantly, we have seen motivated the development of programming languages to greater support artistic expression.

From the embodied view we have taken, it would seem useful to integrate time and space further into programming languages. In practice integrating time can mean on one hand including temporal representations in core language semantics, and on the other uniting development time with execution time, as we have seen with interactive programming. Temporal semantics and interactive programming both already feature strongly in some programming languages for the arts, as we saw in §6, but how about analogous developments in integrating space into the semantics and activity of programming? This is a less well understood area requiring further research, but it would seem that novel approaches to the integration of computational geometry and perceptual models such as computer vision into programming language could serve artists well. By harnessing and extending research into visual programming languages, this could extend to notation, taking for example the ReacTable as inspiration [Jordà et al., 2007].

We began with Paul Klee, a painter whose production was limited by his two hands. The artist-programmer is not so limited, but shares what Klee called his limitation of reception, by the "limitations of the perceiving eye". This is perhaps a limitation to be expanded but not overcome, rather celebrated and fully explored using all we have, including our new computer languages. We have characterised a bricolage approach to artistic programming as an embodied, creative feedback loop. This places the programmer close to their work, grounding symbolic computation in orientational and temporal metaphors of their human experience. However the computer interpreter extends the programmer's abilities beyond their own imagination, making unexpected results likely, leading the programmer to new creative possibilities.

Bibliography

Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, first edition, August 1977. ISBN 0195019199.

Alan Blackwell and Nick Collins. The programming language as a musical instrument. In *Proceedings of PPIG05*. University of Sussex, 2005.

Alan F. Blackwell. Metaphors we program by: Space, action and society in java. In *Proceedings of the Psychology of Programming Interest Group 2006*, 2006a.

Alan F. Blackwell. The reification of metaphor as a design tool. *ACM Trans. Comput.-Hum. Interact.*, 13(4):490–530, December 2006b. ISSN 1073-0516. doi: 10.1145/1188816.1188820.

Margaret A. Boden. *The Creative Mind: Myths and Mechanisms*. Routledge, 2 edition, November 2003. ISBN 0415314534.

Andy Clark. *Supersizing the Mind: Embodiment, Action, and Cognitive Extension (Philosophy of Mind Series)*. OUP USA, November 2008. ISBN 0195333217.

Nick Collins, Alex McLean, Julian Rohrhuber, and Adrian Ward. Live coding in laptop performance. *Organised Sound*, 8(03):321–330, 2003. doi: 10.1017/S135577180300030X.

Edsger W. Dijkstra. On the cruelty of really teaching computing science. 1988.

Conal Elliott. Push-pull functional reactive programming. In *Haskell Symposium*, 2009.

Steven A. Finney. Real-time data collection in linux: A case study. *Behavior Research Methods, Instruments, & Computers*, 33(2):167–173, May 2001.

Jamie Forth, Alex McLean, and Geraint Wiggins. Musical creativity on the conceptual level. In *IJWCC 2008*, 2008.

Peter Gärdenfors. *Conceptual Spaces: The Geometry of Thought*. The MIT Press, March 2000. ISBN 0262071991.

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.

Paul Klee. *Pedagogical sketchbook*. Faber and Faber, 1953.

George Lakoff and Mark Johnson. *Metaphors We Live By*. University of Chicago Press, first edition edition, April 1980. ISBN 0226468011.

Edward A. Lee. Computing needs time. *Commun. ACM*, 52(5):70–79, 2009. ISSN 0001-0782. doi: 10.1145/1506409.1506426.

Alex McLean and Geraint Wiggins. Petrol: Reactive pattern language for improvised music. In *Proceedings of the International Computer Music Conference*, June 2010.

Gregory L. Murphy. *The Big Book of Concepts (Bradford Books)*. The MIT Press, August 2002. ISBN 0262632993.

Marian Petre and Alan F. Blackwell. Mental imagery in program design and visual programming. *International Journal of Human-Computer Studies*, 51:7–30, 1999.

M. Puckette. The patcher. In *Proceedings of International Computer Music Conference*, 1988.

Julian Rohrhuber, Alberto de Campo, and Renate Wieser. Algorithms today: Notes on language design for just in time programming. In *Proceedings of the 2005 International Computer Music Conference*, 2005.

Sherry Turkle. *The Second Self: Computers and the Human Spirit, Twentieth Anniversary Edition*. The MIT Press, 20 anv edition, July 2005. ISBN 0262701111.

Sherry Turkle and Seymour Papert. Epistemological pluralism: Styles and voices within the computer culture. *Signs*, 16(1):128–157, 1990. ISSN 00979740. doi: 10.2307/3174610.

Sherry Turkle and Seymour Papert. Epistemological pluralism and the revaluation of the concrete. *Journal of Mathematical Behavior*, 11(1):3–33, March 1992.

Adrian Ward, Julian Rohrhuber, Fredrik Olofsson, Alex McLean, Dave Griffiths, Nick Collins, and Amy Alexander. Live algorithm programming and a temporary organisation for its promotion. In Olga Goriunova and Alexei Shulgin, editors, *read_me — Software Art and Cultures*, 2004.

G. A. Wiggins. A preliminary framework for description, analysis and comparison of creative systems. *Journal of Knowledge Based Systems*, 2006a.

G. A. Wiggins. Searching for computational creativity. *New Generation Computing*, 24(3): 209–222, 2006b.