Chapter 9
Computer Programming in the Creative Arts

Alex McLean and Geraint Wiggins

Abstract Computer programming is central to the digital arts, and is a comparatively new creative activity. We take an anthropocentric view of computer programming in the arts, examining how the creative process has been extended to include the authorship and execution of algorithms. The role of human perception in this process is a focus, contrasted and ultimately combined with a more usual linguistic view of programming. Practical impacts on the notation of programs in the arts are highlighted, both in terms of space and time, marking out this new domain for programming language design.

9.1 Introduction

Computer programming for the arts is a subject laden with misconceptions and far-flung claims. The perennial question of *authorship* is always with us: if a computer program outputs art, who has made it, the human or the machine? Positions on creativity through computer programming tend towards opposite poles, with outright denials at one end and outlandish claims at the other. The present contribution looks for clarity through a human-centric view of programming as a key activity behind computer art. We view the artist-programmer as engaged in an inner human relationship between perception, cognition and computation, and relate this to the notation and operation of their algorithms.

The history of computation is embedded in the history of humankind. Computation did not arrive with the machine: it is something that humans do. We did not invent computers: we invented machines to help us compute. Indeed, before the arrival of mechanical computers, "*computer*" was a job title for a human employed

A. McLean (✉)
Interdisciplinary Centre for Scientific Research in Music (ICSRiM), University of Leeds, Leeds, LS2 9JT, UK
e-mail: alex@slab.org

G. Wiggins
School of Electronic Engineering and Computer Science, Queen Mary, University of London, E1 4NS, London, UK
e-mail: geraint.wiggins@eecs.qmul.ac.uk

J. McCormack, M. d'Inverno (eds.), *Computers and Creativity*,
DOI 10.1007/978-3-642-31727-9_9,

to carry out calculations. In principle, these workers could compute anything that modern digital computers can, given enough pencils, paper and time.

The textile industry saw the first programmable machine to reach wide use: the head of the Jacquard loom, a technology still used today. Long strips of card are fed into the Jacquard head, which reads patterns punched into the card to guide intricate patterning of weaves. The Jacquard head does not itself compute, but was much admired by Charles Babbage, inspiring work on his mechanical *analytical engine* (Essinger 2004), the first conception of a programmable universal computer. Although Babbage did not succeed in building the analytical engine, his design includes a similar card input mechanism to the Jacquard head, but with punched patterns describing abstract calculations rather than textile weaves.

This early computer technology was later met with theoretical work in mathematics, such as Church's lambda calculus (Church 1941) and the Turing machine (Turing 1992, orig. 1947), which seeded the new field of computer science. Computer programmers may be exposed to these theoretical roots through their education, having great impact on their craft. As it is now practised, however, computer programming is far from a pure discipline, with influences including linguistics, engineering and architecture, as well as mathematics.

From these early beginnings programmers have pulled themselves up by their bootstraps, creating languages within languages in which great hierarchies of interacting systems are expressed. Much of this activity has been towards military, business or scientific ends. However, there are numerous examples of alternative programmer subcultures forming around fringe activity without obvious practical application. The Hacker culture at MIT was an early example (Levy 2002), a group of male model-railway enthusiasts and phone network hackers who dedicated their lives to exploring the possibilities of new computers, under the pay of the military. Many other programming cultures have since flourished. Particularly strong and long-lived is the *demoscene*, a youth culture engaged in pushing computer animation to the limits of available hardware, using novel algorithmic techniques to dazzling ends. The demoscene spans much of the globe but is particularly strong in Nordic countries, hosting annual meetings with thousands of participants (Polgár 2005).

Another, perhaps looser, programmer culture is that of Esoteric Programming Languages or *esolangs*, which Wikipedia defines as "programming language(s) designed as a test of the boundaries of computer programming language design, as a proof of concept, or as a joke". By pushing the boundaries of programming, esolangs provide insight into the constraints of mainstream programming languages. For example, *Piet* is a language notated with fluctuations of colour over a two dimensional matrix. Programs are generally parsed as one dimensional sequences, and colour is generally *secondary notation* (Blackwell and Green 2002) rather than primary syntax. Piet programs, such as that shown in Fig. 9.1, intentionally resemble abstract art, the language itself named after the modernist painter Piet Mondrian. We return to secondary notation, as well as practical use of two dimensional syntax in Sect. 9.4.

Members of the demoscene and esolang cultures do not necessarily self-identify as artists. However, early on, communities of experimental artists looking for new

Fig. 9.1 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 license

means of expression grew around computers as soon as access could be gained. In Great Britain, interest during the 1960s grew into the formation of the Computer Arts Society (CAS)[1] (Brown et al. 2009). However after a creative boom CAS entered a period of dormancy in the mid-1980s, perhaps drowned out by extensive commercial growth in the computer industry at that time. CAS has, however, been revived in more recent years, encouraged by a major resurgence of software as a medium for the arts. This has seen a wealth of new programming environments designed for artists and musicians, such as *Processing* (Reas and Fry 2007), *SuperCollider* (McCartney 2002), *ChucK* (Wang and Cook 2004), *VVVV* (http://vvvv.org) and *OpenFrameworks* (openframeworks.cc), joining more established environments such as the Patcher languages (Puckette 1988), *PureData* and *Max*. These have gained enthusiastic adoption outside a traditional base focused on academic institutions, and have proved useful for teaching the conceptual visualisation required to program computers.

Several artist-programmers have made their own, novel languages in which to make their art. These often seem like esoteric languages that have found practical application. For example unique representations of time are central features of ChucK and SuperCollider. Programming languages have themselves been exhibited as works of art, such as the *Al-Jazari* music programming environment shown in Fig. 9.2 (McLean et al. 2010). Programming languages made for artists have created new and emerging approaches to language design. This is not just a matter of technical achievement, but brings important psychological issues 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 an artist.

[1] www.computer-arts-society.org.

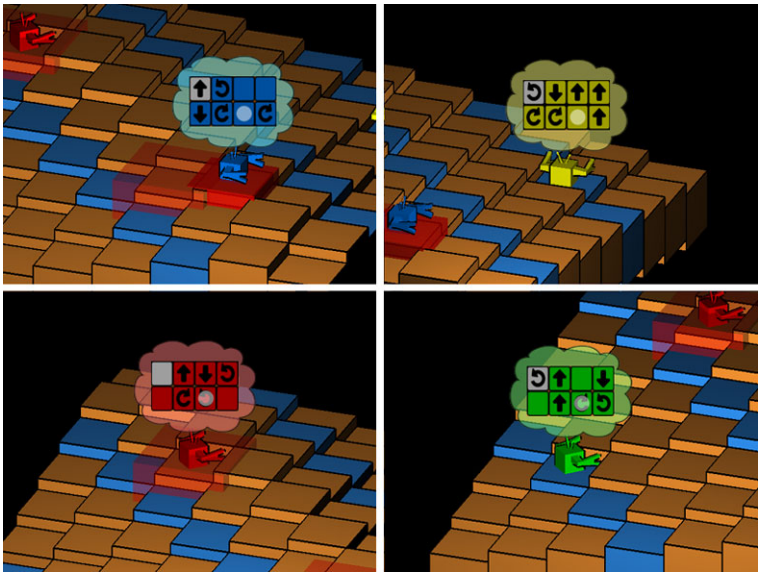

Fig. 9.2 The robots of the Al-Jazari language by Dave Griffiths (McLean et al. 2010). Each robot has a thought bubble containing a small program, edited through a game pad

9.2 Creative Processes

The painter Paul Klee 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**. (Klee 1953, p. 33, original emphasis)

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 text representing 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 (after Lévi-Strauss 1968) 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 mid-course 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";

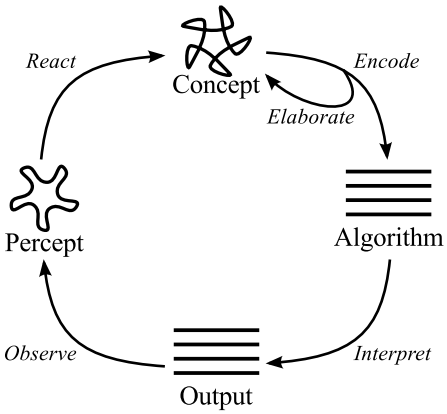

Fig. 9.3 The process of action and reaction in bricolage programming

> for bricoleurs, it is more like a conversation than a monologue. (Turkle and Papert 1990, p. 136)

This concept of bricolage accords with Klee's account, and is also strongly related to that of the *reflective practice* (Schon 1984). This distinguishes the normal conception of knowledge, as gained through study of theory, from that which is learnt, applied and reflected upon while "in the work". Reflective practice has strong influences in professional training, particularly in the educational and medical fields. This suggests that the present discussion could have relevance beyond our focus on the arts.

Although Turkle and Papert address gender issues in computer education, this quote should not be misread as dividing all programmers into two types; while associating bricolage with feminine and planning with male traits (although note Blackwell 2006a), they are careful to state 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 activity, 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 their creative process as a whole, and we seek to understand how in the following.

9.2.1 Creative Process of Bricolage

Figure 9.3 characterises 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. Creative feedback loops are far from unique to programming, but the addition of the algorithmic component 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.

The lack of forward planning in bricolage programming means the feedback loop in Fig. 9.3 is self-guided, possibly leading the programmer away from their initial motivation. This straying is likely, as the possibility for surprise is high, particularly when shifting from the inner loop of implementation to the outer loop of perception. The output of a generative art process is rarely exactly what we intended, and we will later argue in Sect. 9.5 that this possibility of surprise is an important contribution to creativity.

Representations in the computer and the mind 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 range 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 both discrete and continuous representations which underlie cognition (Paivio 1990).

There is something curious about how the programmer's creative process spawns a second, computational one. In an apparent trade-off, the computational process is lacking in the broad 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 Sect. 9.5, after first looking more broadly at how we relate programming to human experience, and related issues of representation.

9.3 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 ground programming language in everyday reasoning. Petre and Blackwell (1999) gave subjects programming tasks, and asked them to introspect upon their imagination

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. 9.4 Conceptual metaphors derived from analysis of Java library documentation by Blackwell (2006b). Program components are described metaphorically as actors with beliefs and intentions, rather than mechanical imperative or mathematical declarative models

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 (2006b) 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 Fig. 9.4. 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.

It would seem, then, that programmers understand the structure and operation of their programs by metaphorical relation to their experience as a human. Indeed the feedback loop described in Sect. 9.2 is by nature anthropomorphic; by embedding the development of an algorithm in a human creative process, the algorithm itself becomes a human expression. Dijkstra 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 1988, p. 22)

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. He 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 (Lakoff and Johnson 1980). 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.

9.4 Symbols and Space

We now turn our attention to how the components of the bricolage programming process shown in Fig. 9.3 are represented, in order to ground understanding of how they may interrelate. Building upon the anthropocentric view taken above, we propose that in bricolage programming, the human cognitive representation of programs centres around perception. Perception results in a low-dimensional representation of sensory input, giving us a somewhat coherent, spatial view of our environment. By spatial, we do not merely mean "in terms of physical objects"; rather, we speak 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, triggering a process we call observation. Recall the programmer from Sect. 9.2, who saw something not represented in the algorithm or even in its output, but only in their own perception of the output; observation is itself a creative act.

The remaining component to be dealt with from Fig. 9.3 is that of programmers' concepts. A concept is "a mental representation of a class of things" (Murphy 2002, p. 5). Figure 9.3 shows concepts mediating between spatial perception and discrete 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. The expressive power of orientational metaphors is that it structures concepts not in terms of one another, but in terms of the orientation of the physical body. These metaphors allow concepts to be related to one another as part of a broad, largely coherent system.

Returning to Fig. 9.4, showing programming metaphors in the Java language, we find the whole class of orientational metaphors described as a single metaphor PROGRAMS OPERATE IN A SPATIAL WORLD WITH CONTAINMENT AND EXTENT. In line with CMT, we suggest this is a major understatement, that orientational metaphors structure the understanding of the majority of fundamental concepts. For example, a preliminary examination leads us to hypothesise that orientational metaphors such as ABSTRACTION IS UP and PROGRESS IS FORWARD would be consistent with this corpus, but further work is required.

Gärdenfors (2000) formalises orientational metaphor by further proposing that the semantic meanings of concepts, and the metaphorical relationships between them are represented as geometrical properties and relationships. Gärdenfors posits that concepts themselves are represented by geometric regions of low dimensional spaces, defined by quality dimensions. These dimensions are 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 linguistic 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 linguistic representation; they are distinct but support one another. This is clear in computing, where hardware exists in our world of continuous space, but thanks to reliable electronics, conjures up a world of discrete computation. As we noted in the introduction, humans are able to conjure up this world too, 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 discrete environment to conduct abstract thought and open channels of communication. On the other hand, a piece of computer software is able to simulate spatial environments, perhaps to host a game world or guide robotic movements, both of which may include some kind of model of human perception.

A related theory lending support to this view is that of *Dual Coding,* developed through rigorous empirical research by Paivio (1990). Humans have a capacity to simultaneously attend to both the discrete codes of language and the analogue codes of imagery. We are also able to reason by invoking quasi-perceptual states, for example by performing mental rotation in shape matching tasks (Shepard and Metzler 1971). Through studying such behaviour Paivio (1990) concludes that humans have a dual system of symbolic representation; an analogue system for relating to modes of perception, and a discrete system for the arbitrary, discrete codes of language. These systems are distinct but interrelate, with "high imagers" being those with high integration between their linguistic and quasi-perceptual symbolic systems (Vogel 2003).

Returning to our theme of programming, the above theories lead us to question the role of continuous representation in computer language. Computer language operates in the domain of abstraction and communication but in general does not at base include spatial semantics. Do programmers simply switch off a whole channel of perception to focus only on the discrete representation of code? It would appear not. In fact, spatial layout is an important feature of *secondary notation* in all mainstream programming languages (Blackwell and Green 2002), which generally allow programmers to add white-space to their code freely with little or no syntactical meaning. Programmers use this freedom to arrange their code so that geometrical features may relate its structure at a glance. That programmers need to use spatial layout as a crutch while composing discrete 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 assert that concordant with Dual Coding theory, the linguistic work of programming is supported by spatial reasoning, with secondary notation helping bridge the divide.

There are few examples of spatial arrangement being part of primary syntax. In the large majority of mainstream programming languages geometric syntax does not go beyond one-dimensional adjacency, although in the Python and Haskell languages statements are grouped according to two dimensional rules of indentation. Even visual programming languages, such as the Patcher Languages mentioned in Sect. 9.1, generally do not take spatial arrangement into account (execution order in Max is given by right-left ordering, but the same can be said of 'non-visual' programming languages).

As we noted in Sect. 9.1, the study of "Programming Languages for the Arts" is pushing the boundaries of programming notation, and geometrical syntax is no exception. There are several compelling examples of geometry used in the syntax of languages for music, often commercial projects emerging from academic research. The *ReacTable* (Jordà et al. 2005) is a tangible, multi-user interface, where blocks imprinted with computer readable symbols are placed on a circular display surface (Fig. 9.5). We consider the ReacTable as a programming language environment, although it is not presented as such by its creators. Each symbol represents a sound synthesis function, with a synthesis graph formed based upon the pairwise proximity of the symbols. Relative proximity and orientation of connected symbols are used as parameters modifying the operation of synthesis nodes. Figure 9.6 shows a

Fig. 9.5 The ReacTable (Jordà et al. 2005): a tangible interface for live music, presented here as a programming language environment

screenshot of *Text*, a visual language inspired by the ReacTable and based upon the pure functional Haskell programming language. In Text, functions and values may be placed freely on the page, and those with compatible types are automatically connected together, closest first. Functions are *curried*, allowing terse composition of higher order functions. Text could in theory be used for general programming, but is designed for improvising live music, using an underlying musical pattern library (McLean and Wiggins 2010b). A rather different approach to spatial syntax is taken by *Nodal*, where distance between symbols represents elapsed time during interpretation (McCormack and McIlwain 2011). The result is a control flow graph where time relationships in musical structure can be easily seen and manipulated as spatial relationships.[2] In all of these examples, the graphs may be changed while they are executed, allowing interactive composition and indeed live improvisation of the like examined in Sect. 9.6.

An important assertion within CMT is that a conceptual system of semantic meaning exists within an individual, and not as direct reference to 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 programming metaphors discussed in Sect. 9.3, as well as the general failure of attempts at designing fixed metaphors into computer interfaces (Blackwell 2006c). Each programmer has a different set of worldly interests and experiences, and so establishes different

[2]This space/time syntax can also be seen in Al-Jazari mentioned earlier and shown in Fig. 9.2.

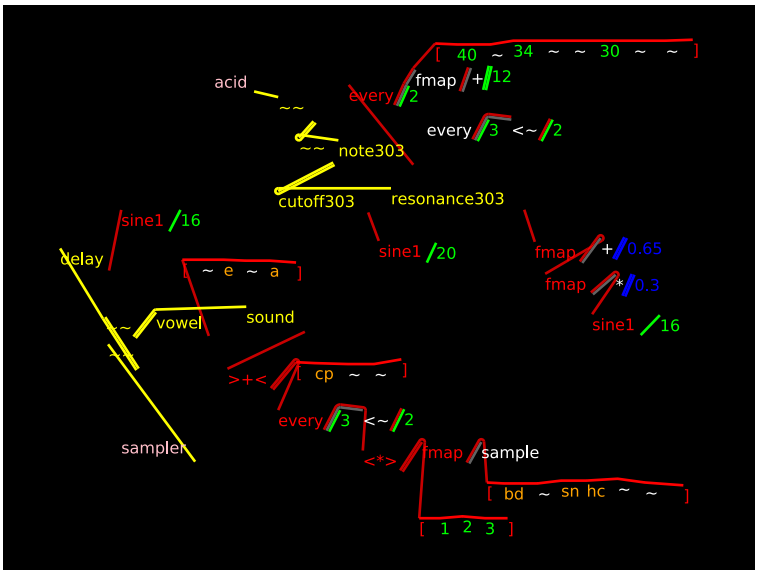

Fig. 9.6 Text, a visual programming language designed for improvised performance of electronic dance music. Functions automatically connect, according to their distance and type compatibility

metaphorical systems to support their programming activities. However, by building orientational and spatial metaphors into programming notation, such as TIME IS DISTANCE, PROXIMITY IS CONNECTIVITY and ORIENTATION IS EXTENT, universal bodily relationships are employed. This results in metaphors that are more readily understood, employing general cognitive resources to artistic expression.

9.5 Components of Creativity

We now have grounds to formally characterise how the creative process operates in 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, using the quasi-Platonic idea, common in AI, that there is an effective duality between exploration of an extant range of items, that conform to rules, and construction of new items according to those rules, in a context where the extent of the space is unknown. Within the CSF, a creative search has three key aspects: the conceptual *search space* itself, *traversal* of the space and *evaluation* of concepts found in the space. In other words, creativity requires somewhere to search, a manner of searching, and a means to judge what you find. However, creative behaviour may make use of introspection, self-modification and need boundaries to be broken. That is, the constraints of search space, traversal and evaluation are not fixed, but are examined, challenged and modified by the creative agent following

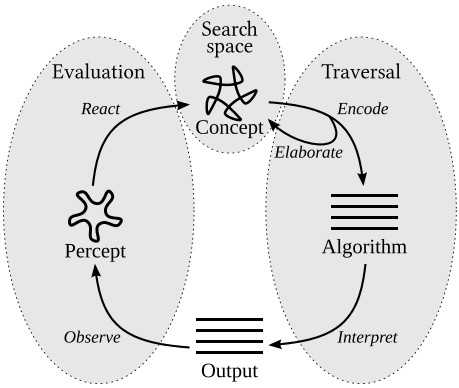

Fig. 9.7 The process of action and reaction in bricolage programming from Fig. 9.3, showing the three components of the Creative Systems Framework, namely search space, traversal strategy and evaluation

(and defined by) them. The CSF supplies tests for particular kinds of *aberration* from the expected conceptual space and suggests approaches to addressing them.

Again using the terminology of Gärdenfors (2000), the search spaces of the CSF are themselves concepts, defining regions in a universal space defined by quality dimensions. Thus, transformational creativity is a geometrical transformation of these regions, motivated by a process of searching through and beyond them; crucially, the search space is not closed. As we will see, this means that a creative agent may creatively push beyond the boundaries of the search. While acknowledging that creative search may operate over linguistic search spaces, we focus on geometric spaces grounded in perception. This follows our focus on artistic bricolage (Sect. 9.2), which revolves around perception. For an approach unifying linguistic and geometric spaces see Forth et al. (2010).

We may now clarify the bricolage programming process introduced in Sect. 9.2.1 within the CSF. As shown in Fig. 9.7, the search space defines the programmer's concept, being their current artistic focus structured by learnt techniques and conventions. The traversal strategy is the process of attempting to generate part of the concept by encoding it as an algorithm, which is then interpreted by the computer. Finally, evaluation is a perceptual process in reaction to the output.

In Sect. 9.2, 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.[3] The above makes clear what we claim is being externalised: part of the traversal strategy. The programmer's concept motivates a development of the traversal strategy, encoded as a computer program, but the programmer does not necessarily have the cognitive ability to fully evaluate it. That task is taken on by the interpreter running on a computer system, meaning that traversal encompasses both encoding by the human and interpretation by the computer.

The traversal strategy 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 many

[3] See also Chap. 14 by Bown.

classes of programming language. Each design pattern identifies a kind of problem, and describes a generalised structure towards a solution.[4]

The creative process is guided by the programmer's concept 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 traversal extends outside the bounds of the search space. If the discovered conceptual instance is valued, then the search space may be extended to include it. If, however, it is not valued, then the traversal strategy may be modified to avoid similar instances in the future.

Because the traversal strategy of a programmer includes external notation and computation, they are likely to be less successful in writing software that meets their preconceptions, or in other words more successful in being surprised by the results. A creative process that includes external computation will follow less predictable path as a result. Nonetheless the process has the focus of a search space, and is guided by value in relation to a rich perceptual framework, and so while unpredictable, this influence is far from random, being meaningful interplay between human/computer language and human perceptual experience. The human concepts and algorithm are continually transformed in respect to one another, and to perceptual affect, 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 motivation for the field of computational creativity in grounding an artificial creative agent in its environment. This is done by acquiring computational models of perception sufficient for the agent to both evaluate its own works and structure its conceptual system. Then the agent would 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 coherent within that culture. For now, however, this is wishful thinking, and we must accept generative computer programs which extend human creativity, but are not creative agents in their own right.

9.6 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

[4]This structural heuristic approach to problem solving is inspired by work in the field of urban design (Alexander et al. 1977).

> 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. Dijkstra might argue that considering computer programs as operating in time at all, rather than as entirely abstract logic, is itself a form of the anthropomorphism examined in Sect. 9.3. 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. This could be a state of optimum experience, the "flow" investigated by Csikszentmihalyi where "duration of time is altered; hours pass by in minutes, and minutes can stretch out to seem like hours" (Csikszentmihalyi 2008, p. 49). Perhaps in this state 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, anthropocentric direction of software entering human time. While programmers can appear detached from "physical" time, there are ways in which the timelines of program development and operation 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 of accuracy and latency. 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, as 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 sub-field of computer science known as *reactive programming* (Elliott 2009), with applications emerging in music (McLean and Wiggins 2010a).

9.6.1 Interactive Programming

Interactive programming allows a programmer to examine an algorithm while it is interpreted, taking on live changes without restarts. This unites the time flow of a

program with that of its development, using dynamic interpretation or compilation. 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 Figs. 9.3 and 9.7 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 through its journey of development from nothing to complex algorithm, generating continuously changing musical or visual form along the way. This is bricolage programming taken to a logical and artistic conclusion.

9.7 Conclusion

What we have discussed provides strong motivation for addressing the concerns of artist-programmers. These include concerns of workflow, where elapsed time between source code edits and program output slows the creative process. Concerns of programming environment are also important, which should be optimised for the presentation of shorter programs in their entirety to support bricolage programming, rather than hierarchical views of larger codebases. Perhaps most importantly, we have seen motivation for the development of new programming languages, pushing the boundaries 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 Sect. 9.6, but how about analogous developments in integrating geometric relationships into the semantics and activity of programming? It would seem the approaches shown in Nodal, the ReacTable and Text described in Sect. 9.1 are showing the way towards greater integration of computational geometry and perceptual models into programming language. This is already serving artists well, and could become a new focus for visual programming language research.

We began with Paul Klee, a painter whose production was limited by his two hands. The artist-programmer is limited differently to the painter, 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: 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 discrete 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.

Acknowledgements Alex McLean was supported by a Doctoral grant awarded by the UK EPSRC.

References

Alexander, C., Ishikawa, S., & Silverstein, M. (1977). *A pattern language: towns, buildings, construction* (1st ed.) London: Oxford University Press.

Blackwell, A. (2006a). Gender in domestic programming: from bricolage to séances d'essayage. In *CHI workshop on end user software engineering*.

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

Blackwell, A. F. (2006b). Metaphors we program by: space, action and society in java. In *Proceedings of the psychology of programming interest group 2006*.

Blackwell, A. F. (2006c). The reification of metaphor as a design tool. *ACM Transactions on Computer-Human Interaction*, *13*(4), 490–530.

Blackwell, A., & Green, T. (2002). *Notational systems—the cognitive dimensions of notations framework* (pp. 103–134). San Mateo: Morgan Kaufmann.

Boden, M. A. (2003). *The creative mind: myths and mechanisms* (2nd ed.). London: Routledge.

Brown, P., Gere, C., Lambert, N., & Mason, C. (Eds.) (2009). *White heat cold logic: British computer art 1960–1980*. *Leonardo books*. Cambridge: MIT Press.

Church, A. (1941). *The calculi of lambda conversion*. Princeton: Princeton University Press.

Clark, A. (2008). *Supersizing the mind: embodiment, action, and cognitive extension*. *Philosophy of mind series*. OUP USA.

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

Csikszentmihalyi, M. (2008). *Flow: the psychology of optimal experience*. HarperCollins eBooks.

Dijkstra, E. W. (1988). *On the cruelty of really teaching computing science (EWD-1036)*. E.W. Dijkstra Archive. Center for American History, University of Texas at Austin.

Elliott, C. (2009). Push-pull functional reactive programming. In *Haskell symposium*.

Essinger, J. (2004). *Jacquard's web: how a Hand-Loom led to the birth of the information age* (1st ed.). London: Oxford University Press.

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

Forth, J., McLean, A., & Wiggins, G. (2008). Musical creativity on the conceptual level. In *IJWCC 2008*.

Forth, J., Wiggins, G., & McLean, A. (2010). Unifying conceptual spaces: concept formation in musical creative systems. *Minds and Machines*, *20*(4), 503–532.

Gärdenfors, P. (2000). *Conceptual spaces: the geometry of thought*. Cambridge: MIT Press.

Jordà, S., Kaltenbrunner, M., Geiger, G., & Bencina, R. (2005). The reacTable. In *Proceedings of the international computer music conference (ICMC 2005)* (pp. 579–582).

Klee, P. (1953). *Pedagogical sketchbook*. London: Faber and Faber.

Lakoff, G., & Johnson, M. (1980). *Metaphors we live by* (1st ed.). Chicago: University of Chicago Press.

Lee, E. A. (2009). Computing needs time. *Communications of the ACM*, *52*(5), 70–79.

Lévi-Strauss, C. (1968). *The savage mind*. *Nature of human society*. Chicago: University of Chicago Press.

Levy, S. (2002). *Hackers: heroes of the computer revolution*. Baltimore: Penguin Putnam.

McCartney, J. (2002). Rethinking the computer music language: SuperCollider. *Computer Music Journal*, *26*(4), 61–68.

McCormack, J., & McIlwain, P. (2011). Generative composition with nodal. In E. R. Miranda (Ed.), *A-Life for music: music and computer models of living systems, computer music and digital audio* (pp. 99–113). A-R Editions.

McLean, A., Griffiths, D., Collins, N., & Wiggins, G. (2010). Visualisation of live code. In *Electronic visualisation and the arts*, London, 2010.

McLean, A., & Wiggins, G. (2010a). Petrol: reactive pattern language for improvised music. In *Proceedings of the international computer music conference*.

McLean, A., & Wiggins, G. (2010b). Tidal—pattern language for the live coding of music. In *Proceedings of the 7th sound and music computing conference*.

Murphy, G. L. (2002). *The big book of concepts*. *Bradford books*. Cambridge: MIT Press.

Paivio, A. (1990). *Mental representations: a dual coding approach*. *Oxford psychology series* (new ed.). London: Oxford University Press.

Petre, M., & Blackwell, A. F. (1999). Mental imagery in program design and visual programming. *International Journal of Human-Computer Studies*, *51*, 7–30.

Polgár, T. (2005). *Freax*. CSW-Verlag.

Puckette, M. (1988). The patcher. In *Proceedings of international computer music conference*.

Reas, C., & Fry, B. (2007). *Processing: a programming handbook for visual designers and artists*. Cambridge: MIT Press.

Rohrhuber, J., de Campo, A., & Wieser, R. (2005). Algorithms today: notes on language design for just in time programming. In *Proceedings of the 2005 international computer music conference*.

Schon, D. A. (1984). The reflective practitioner: how professionals think. In *Action* (1st ed.). New York: Basic Books.

Shepard, R. N., & Metzler, J. (1971). Mental rotation of three-dimensional objects. *Science (New York, N.Y.)*, *171*(972), 701–703.

Turing, A. M. (1992). Intelligent machinery. Report, national physics laboratory. In D. C. Ince (Ed.), *Collected works of A. M. Turing: mechanical intelligence* (pp. 107–127). Amsterdam: Elsevier.

Turkle, S. (2005). *The second self: computers and the human spirit* (20 anv. ed.). Cambridge: MIT Press.

Turkle, S., & Papert, S. (1990). Epistemological pluralism: styles and voices within the computer culture. *Signs*, *16*(1), 128–157.

Turkle, S., & Papert, S. (1992). Epistemological pluralism and the revaluation of the concrete. *Journal of Mathematical Behavior*, *11*(1), 3–33.

Vogel, J. (2003). Cerebral lateralization of spatial abilities: a meta-analysis. *Brain and Cognition*, *52*(2), 197–204.

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

Ward, A., Rohrhuber, J., Olofsson, F., McLean, A., Griffiths, D., Collins, N., & Alexander, A. (2004). Live algorithm programming and a temporary organisation for its promotion. In O. Goriunova & A. Shulgin (Eds.), *read_me—software art and cultures*.

Wiggins, G. A. (2006a). A preliminary framework for description, analysis and comparison of creative systems. *Journal of Knowledge Based Systems*, *19*, 449–458.

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