CODING PRAXIS: RECONSIDERING THE AESTHETICS OF CODE

GEOFF COX, ALEX MCLEAN,

ADRIAN WARD

IN 'reconsidering the aesthetics of code', we hope to reflect upon an earlier essay,[1] and to extend its remit in understanding code as performative: that which both performs and is performed.

The previous paper 'The Aesthetics of Generative Code' (2000) drew an analogy between poetry and code. Appreciation of poetry may come from reading or experiencing a live spoken performance. Similarly, code may have aesthetic value in both its written form and its execution. The paper argued that any separation of code and the resultant actions would simply limit the aesthetic experience (otherwise based purely on the sense apparatus), and ultimately limit the study of these 'generative' forms (that should also engage with the technical apparatus itself). Speech and its representation in writing together form a language that we appreciate as poetry. In the essay we speculated whether code could be seen to work in a similar way?

Put simply, this emphasises that art-orientated programming needs to acknowledge the conditions of its own making—its poesis. This requires both a technical and cultural impulse reflecting current thinking in the critical discourse around 'software art'. For instance, it has become a truism to emphasise that code is not merely functional but can have poetic qualities, and political significance. Florian Cramer, amongst others, provides many examples of poetic approaches to programming: from dada poetry to the conceptual art tradition, from perl poetry, code slang, 'viral scripting, in-code recursions and ironies'.[2] This is nothing new as he

acknowledges. Back in 1981 Donald Knuth pointed to the added value of programming as not merely 'economically and scientifically rewarding, but also [...] an aesthetic experience much like composing poetry or music'.[3] In such examples, the code is the material—most clearly likened to the materiality of language—leading to the idea of software as potential literature.[4] In this formulation, the formal qualities of code cannot be separated from its broader discursive framework.

reconsidering the terms

This paper aims to address these ideas in the light of our current thinking and practice. Clearly, in writing any essay, like writing code, a properly critical approach should not simply fix data and ideas. History is inherently unstable, contradictory, dynamic and dialectical, and therefore any essay or piece of code is only ever a work in progress and should be subject to active criticism and upgrade.

Even the title itself, 'the aesthetics of generative code', needs qualification. One way of approaching a term like 'generative code' is from 'generative grammar', a linguistic theory proposed by Noam Chomsky (in *Syntactic Structures*, 1972) to refer to deep-seated rules by which language operates.[5] In keeping with this, much recent generative work has been concerned with the linguistic qualities of code work (sometimes called 'code literature' or 'code poetry'). This linguistic approach would seem suitable for us too in emphasising that linguistics needs an abstract system demonstrating competence ('langue'), that generates the concrete event or performance ('parole', to use Saussure's terms). Furthermore, all conventions of writing and reading (even of code) can be said to be part of a set of abstract (coded) systems by which outputs are generated and understood. According to this way of thinking, it is as if the text is relatively autonomous from the act of writing—as if writing writes rather than writers. Evidently, if writing can be seen to be autonomous, it can also be seen to be self-organising or generative. For instance, Calvino's 'How I Wrote One of My Books' referred to his popular book *If on a Winter's Night a Traveller*.[6]

It reads like a mathematical formula, as an algorithmic description of the book's structure—and with this in mind, the subtitle of our previous essay was: 'how I wrote one of my perl scripts'. Codes are essentially closed systems of semiotic elements—like all language codes. The texts which are formulated in these languages (or programs) are 'performative strings of signifiers.'[7]

what is code?

Code often begins as a vague idea of its final form. In its infancy, the process of breaking down the finished product into creatable parts produces new parts, ideas and directions that shape the final form often in some new and unexpected way. Amongst the many techniques for writing software, applying a conceptual framework to the product helps to define the constituent parts. Traditionally, flow charts help to visualise the procedural flow of code. Object-orientated design is a more contemporary approach to rendering the entire product into manageable components. At all times, the programmer retains an overall understanding of the product but can concentrate on the details using these techniques. Perhaps the similarity here to how a piece of music may be constructed is useful in that themes and elements are developed and brought together in order to build a coherent whole—although this analogy may be too simple. Does the composer always imagine their finished piece before it exists? Does the programmer always know the desired result before they launch Emacs?

What this paper suggests is that creative production (whether it be code, or music) is performative, where the potential for change is very much active and dynamic. Mistakes are made, which themselves may lead to further possibilities. Even when the product is commercially-driven and the goals are very much predetermined, the techniques invoked are not prescribed. The programmer is constantly learning new techniques. This however, does not deny the importance of theoretical activity—in fact, it is the very presence of theory that makes the practice applicable.

how does code run?

In the previous paper, we described code as a notation of an internal structure that the computer is executing, expressing ideas, logic, and decisions that operate as an extension of the programmer's intentions. The written form is merely a computer-readable notation of logic, and is a representation of this process. Yet the written code isn't what the computer really executes, since there are many levels of interpreting and compiling and linking taking place. Code is only really understandable within the context of its overall structure—which is what makes it like a language (be it source code or machine code, or even raw bytes). To appreciate this fully we need to 'perceive' the code to grasp what it is we are experiencing and to build an understanding of the code's actions.

Once authored, code is executed. In technical terms, the processor is obeying the instructions given to it and generating activity by deploying automation. In many ways, it is easy to see this step as a solidification of the creative process since the hand of the author is not physically felt. But software itself relies on the deferred action of its author—the code operates on behalf of the programmer, so it is more accurate to consider this as part of a continuing performance. This is especially obvious when interaction exists between the software and a user, and even more so when the coder is the user. Even in business terminology, software performs. However, it is important to recognise that the actions of a piece of code consist of a great deal more than a translation of the intentions of its author. The code is interacting with the user, itself, its environment, and the systems it has access to via the many multi-layered and mediated interfaces that are available to it. The Operating System defines potential activities via APIs, the hardware defines potential functions via machine code, and yet these are implicit and mostly unseen. The performance is thus the result of many components, from a wide range of sources, interacting dynamically. Many of the components are predetermined, but through the combinations of interactions combined with the dynamism and unpredictability of live action, the result is far from fixed as a whole.

The following example illustrates the turmoil of the environment in which code executes. This simple program, called *hot_air_balloon.pl*, burns system resources whenever system load falls below a certain level. It both reacts to and changes the environment in which it runs. All programs share this two-way relationship with the systems they inhabit. All programs load the systems they run within, and demands made by programs cannot always be served by the operating system in which memory capacity and processing cycles are limited.

```
#!/usr/bin/perl
while (1) {
    open(FH,  </proc/loadavg ) or return;
    my ($load_average) = <FH> =~ /([\d\.]+)/;
    close FH;
    if ($load_average < 0.5) {
       my @foo = sort map {rand} (1 .. 1_000_000);
    }
    sleep(1);
}
```

formalist reception

This paper so far describes an approach that runs counter to the tendency to hide the code that lies behind the work. Cramer laments how in much interactive art, the impression is that the viewer makes the work somehow through 'interaction' rather than the complex interactions of processes and code running on the computer behind the scenes—demonstrating both an ignorance of programming and of course of programmers.[8] In the previous paper, this was partly our intention to highlight these relatively hidden aesthetic pleasures of code and coding. Since publication, however, this has been generally received as indicative of an over-concentration of formal concerns at the expense of cultural or social implications. In 'Concepts, Notations, Software, Art', Cramer refers to this as 'software formalism', as distinct from 'software culturalism' to characterise what he

saw at that time as two distinct tendencies.[9] Following this, Erkki Huhtamo is interested too in what he calls 'software art purists' that emphasise the primacy of code as the main artistic material. Huhtamo sees the software art movement as a continuation of a neo-modernist project: 'emphasizing the centrality of the code and the algorithmic approach [...] positing a "hard core" often felt to be lost in the postmodern world'. In this connection our earlier essay is seen to 'fulfill many of the criteria for classical avant-garde movements' as distinct from culturalist approaches that do not fit so easily in the 'modernist straitjacket'[10] — an expression of conceptual madness in other words.

This aspect of the essay's reception has been disappointing, as it was only ever thought to be part of the argument. It was claimed that the ideological aspects (what might be called the 'generative matrix') lay outside the scope of our first paper but that it was necessary for a fuller and sustained criticism. But there is also a common assumption here that modernism is deterministic and an emphasis on code reductive. On the contrary, we might argue that the received view of modernity has stagnated and still might be usefully presented in terms of contradiction appropriate to the examination of code.

To the art historian, formalist concerns make reference to the unfashionable ideas of Clement Greenberg, in which structure and 'pure' form are seen to reveal the essence of the work. To the media art historian, Russian Formalism is perhaps evoked: 'Every work, every novel, tells through its fabric of events the story of its own creation, its own history... the meaning of a work lies in its telling itself, its speaking of its own existence'[11] — every work carries with it its source code in other words. But in this attention to form, there is an implied politics of a work's own making if we extend the reference to other critical modernist practices, such as in the work of Brecht or Benjamin (appropriate to the examination of generative work). For instance, Benjamin recommends that the 'cultural producer' intervene in the production process, in order to transform the apparatus.[12] In other words, it is only through an engage-

ment with the technical apparatus the cultural producer can engage with the relations of production. Clearly this is not pure form (nor simply free nor open) but a decidedly dirty form of critical engagement appropriate to the political task of exposing hidden exploitation in order to change it for the better.

code criticism

Admittedly, there is some danger of the aestheticisation of code at the expense of other factors—analogous to the concerns of Benjamin in calling for a politics of aesthetics rather than an aesthetics of politics (in his artwork essay). This distinction can be applied to the undialectical separation of technical concerns over aesthetic ones. By extension, and in this context, we need to examine art-orientated programming and program-orientated art. Clearly and in unashamedly modernist terms, 'both-and' are preferred over 'either-or' (as Marshall Berman would put it).[13] In this way, perhaps the contradictions involved in the production of code, and the relations of its production, might be somewhat revealed.

There is a risk of making critical claims that contradict the very principles of code as something 'generative'; that is always in progress, and on execution produces unpredictable and contradictory outcomes. It is in a continuous state of 'becoming' (to use Henri Bergson's phrase). The recent book by Matthew Fuller, *Behind the Blip* (2003), is useful in this connection in calling for a criticism of 'software culture' that does not operate at some distance from practice but that takes account of practice. Fuller does this through presenting examples of practices and categories that are not exclusive but simply ideas in progress, briefly summarised as: firstly, 'critical software' designed to undermine normalised understandings of the operations of software itself; secondly, 'social software' developed and changed through social networks of users and programmers, that emerges from a different set of social relations such as those of the 'open source' community; and thirdly, 'speculative software' that reflexively investigates itself, what he calls the 'reinvention of software by its own means... as mutant epistemology'.[14]

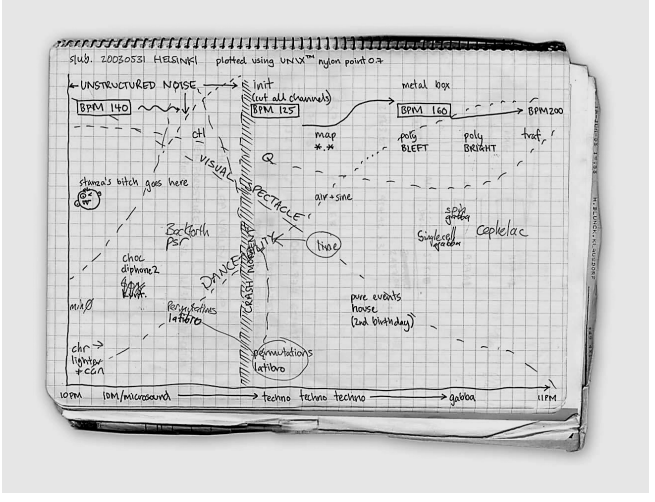
Additionally, any 'theory' of software is itself speculative, and requires an understanding of the complex interactions of processes, undertaking theorization that is 'able to operate on the level of a particular version of a program, a particular file-structure, protocol, sampling algorithm' and so on[15]—moving from the general to the particular in other words. Fuller sees 'the task of such practical and critical work to open these layers up to the opportunity of further assemblage'.[16]

programmer as producer

Thus criticism has to engage with code in its fullest sense. This has parallels in hardware too where the mechanisms and relations of production remain hidden for the most part—presented as deterministic and unchangeable. This can be traced back to the beginnings of the industrial period not least, reflecting a trend to alienate the worker or user from the very processes they are involved in. For William Bowles (1990), this is entirely expected in that craft skills are stolen (or living labour is replaced by dead labour) and ever more sophisticated machines require less and less skill to operate.[17] There is an obvious parallel here in computing and the craft of programming. For Alan Sondheim too: 'every more or less traditional text is codework with invisible residue'[18] of creative labour. Despite surface appearances, the processes involved are decidedly complex and there is a vast amount of expertise invested in whatever operating system is running. For the most part these processes are closed off either by the design of the operating system or by the lack of programming knowledge of the user. To Cramer, it is almost as if graphical software disguises itself as hardware.[19] On the other hand, the Unix command line holds multiple possibilities for transformation and manipulation—combining instruction code and conventional written language into potentially poetic forms. Rather than the readerly properties of a graphical operating system that encourages consumption and hides the code, the command-line operating system of Unix is seen as writerly in terms of openness and encouraging the reader to

become a producer of text or code. This is important for Cramer as it breaks down the false distinction between the writing and the tool with which the writing is produced, and in terms of the computer between code and data. Coding requires human intervention and full access to the means of production. In this formulation, the human subject gains agency (the power to act), as one who assembles the apparatus as much as is assembled by it.

Code requires speculation; programmers execute it in their heads as they write it. Since a programmer's task is to develop a system which not only uses variables but is variable itself, it is necessary for the programmer to be able to know, or at least perceive the states through which a piece of code moves and how these states inform other operations in order to build a coherent system. A programmer is therefore able to predict and speculate upon how their code will behave in most usual circumstances. As with anything that is authored, the issue of subjectivity is unavoidable, since any particular result can be achieved in many different (and often competing) ways. In this, any sense of improvisation relies on a predictive understanding of complex and generative systems. There is some risk involved and in the case of the generative



music performances of *slub* (aka Alex McLean and Adrian Ward), the intention is to open up what otherwise would seem to be determinate processes of how music is generated.[20] We hear glitches and all.

A live performance that includes authoring code diverges from any perceived determination. Like the initial stages of software development, a performance may be sketched out or planned roughly—the entire performance might be imagined but many of the details are not yet known. Much preparation is done for the start of the performance, but it is only when the performance begins that those details will start to form and inform the performance itself. The guidelines and structures initially developed merely exist as a framework of possibilities, but it is simply not possible at this stage to this as merely a rigid execution of instructions. Without denying the performative elements of a finished piece of code in itself, live-coding is a further development of a performance scenario. This includes the potential for mistakes to positively affect the course of the performance, that no longer expresses control but is open to the vagaries of feedback.

self-modifying code

```
sub bang {
  my $self = shift;
  # Feedback
  $self->code->[3]=~ s/e/ee/;
  $self->modified;
}
sub bang {
  my $self = shift;
  # Feeeeeeedback
  $self->code->[3]=~ s/e/ee/;
  $self->modified;
}
sub bang {
  my $self = shift;
```

```
    # Feeeeeeeeeeeeeeeeeeeeeeedback
    $self->code->[3] =~ s/e/ee/;
    $self->modified;
}
```

http://yaxu.org/words/yaxu/feedback.html

Although code is programmed, it is not simply causal and remains unpredictable. In our example, *feedback.pl*, a text editor is editing a piece of code that has the ability to modify itself when executed. These modifications happen directly to the code being edited in realtime, opening up the possibility for the code to fundamentally modify its own behaviour. Of course, this has major implications upon the act of programming. The programmer must now think not only of what the software will do and how it will interact, but also how it will modify itself and remain functional, and continue to be active. The programmer must take a leap of faith and now consider the code's initial logic as well as be able to follow the code's logic after it has modified itself, and continue to do so indefinitely. In a sense, the software itself must contain an understanding of its own performance, and be able to sustain this performance through its performance. Here, we might speculate on understanding code as embedding both theory (of its own agenda) and practice (of its own action).

Additionally, the speculative distinction between the tool and the machine is broken here. It is clear that some software exists as a tool (as an extension of the human body) and some software exists as a machine (a technique for automation). Coding is the practice of creating either scenario, but a code that reflects upon its own making in this way cannot simply be regarded as a tool—it is reflexive. Nor is it simply a machine, since its actions are highly variable and likely to fail. The similarities between this and the self-destructive nature of many performances cannot go unnoticed. Self-modifying code blatantly breaks the determinism of code and makes it explicit.

speculating on self-critical code

In addressing the idea of the limits of the aesthetics of code, the suggestion is that some of the hidden oppositions between the intellectual and physical division of labour involved in programming might be revealed as a consequence. In this paper, we aim to make a further analogy to the dialectical relationship of theory to practice—in the lexicon of critical theory known as 'praxis'. Praxis is a self-creating action informed by theory, and therefore thoroughly active and dynamic. Is the analogy of this concept productive for thinking about the role of code beyond its functional role, and its implications for offering a program(me) of action?

In our example, code literally performs, is thoroughly 'write-able', and reflects its intrinsic contradictory and potentially disruptive qualities. We maintain that art-orientated programming can be useful in revealing these inherent contradictory tendencies. Adorno says: 'A successful work of art [...] is not one that resolves objective contradictions in a spurious harmony, but one that expresses the idea of harmony negatively by embodying the contradictions, pure and uncompromised, in its innermost structure'.[21] Perhaps 'coding praxis' should be seen as a contradiction in terms, and productively so.

Is it possible to be able to produce code that encapsulates
the possibility of a critical practice in this way?
This is speculative thinking of course—
and probably requires a further
paper to reconsider
the claims
of this
one
.

[1] Geoff Cox, Adrian Ward & Alex McLean, 'The Aesthetics of Generative Code', *Generative Art 00*, international conference, Politecnico di Milano, Italy, 2000, http://www.generative.net/papers/aesthetics/index.html

[2] Florian Cramer, 'Exe.cut[up]able statements: the Insistence of Code', in Gerfried Stocker & Christine Schöpf, eds., *Code— The Language of Our Time*, Ars Electronica, Linz: Hatje Cantz, 2003, p.102

[3] Donald Knuth, *The Art of Computer Programming: Volume 1, Fundamental Algorithms*, (first published 1968), Reading, Massachusetts: Addison-Wesley, 1981: v

[4] This is a reference to *ouvroir de potentiale litterature (OuLiPo)*

[5] Noam Chomsky, *Syntactic Structures* (first published 1957), The Hague: Mouton, 1972; further reading on Chomsky's early work in generative processes, http://www.ifi.unizh.ch/groups/CL/volk/SyntaxVorl/Chomsky.html

[6] Italo Calvino, 'How I Wrote One of My Books', trans. Iain White, in *OuLiPo Laboratory*, London: Atlas, 1995

[7] Findeisen's statement in, 'Some Code to Die for', quoted in Stocker & Schöpf, op. cit., p.74

[8] Florian Cramer, 'Concepts, Notations, Software Art', in *Signwave, Auto-Illustrator Users Guide,* Plymouth: Liquid Press/Spacex, 2002, p.102, http://www.netzliteratur.net/cramer/concepts_notations_software_art.html

[9] Florian Cramer, ibid., p.110

[10] Erkki Huhtamo, 'Web Stalker Seek Aaron: Reflections on Digital Arts, Codes and Coders', in Stocker & Schöpf, op. cit., 2003, p.117

[11] Jakobson, from *Littérature et signification*, quoted in Terence Hawkes, *Structuralism and Semiotics* (first published 1977), London: Methuen, 1986, p.100

[12] Walter Benjamin, 'The Author as Producer' in *Understanding Brecht*, trans. Anna Bostock, London: Verso, 1983, p.98

[13] Marshall Berman, *All That Is Solid Melts Into Air: the Experience of Modernity*, London: Verso, 1999

[14] Matthew Fuller, *Behind the Blip: essays on the Culture of Software*, New York: Autonomedia, 2003, p.30

[15] Fuller, ibid., p.17

[16] Fuller, ibid., p.21

[17] William Bowles, 'The Macintosh Computer: Archetypal Capitalist Machine?' *Retrofuturism* 13, (first written in 1987), 1990, http://psrf.detritus.net/r/13/index.html

[18] Alan Sondheim, 'Notes on Codework', *nettime*, February 11, 2004

[19] Cramer, op. cit., 2003, p.101

[20] Nick Collins, Alex McLean, Julian Rohrhuber, & Adrian Ward, 'Live Coding in Laptop Performance', in *Organised Sound* 8 (3) Cambridge University Press, 2003, p.326; these ideas have been further developed in, Adrian Ward, Julian Rohrhuber, Fredrik Olofsson, Alex McLean, Dave Griffiths, Nick Collins, Amy Alexander, 'Live Algorithm Programming and a Temporary Organisation for its Promotion', 2004, http://www.toplap.org

[21] Adorno, from *Prisms*, 1967, p.32, quoted in, Martin Jay, *The Dialectical Imagination: A History of the Frankfurt School and the Institute of Social Research 1923–1950* (first published 1973), London: University of California Press, 1996, p.179

An earlier version of this paper was presented at the symposium Programmation-Orientee Art, University of Paris: Sorbonne, March 2004.