Digital Creativity
2006, Vol. 17, No. 1, pp. 3–10

Composing along continuums of technology and purpose

Ron Herrema
De Montfort University, UK
RHerrema@dmu.ac.uk

Abstract

It is tempting to see artists who write their own computer code as comprising a circumscribed minority having a peculiar temperament. This article attempts to show that in the current digital world there exist programs that enable artists to move along a continuum of coded and non-coded computer use. In addition, it uses the personal creative experiences of the author to illustrate the value for artists of moving into the coded end of the continuum.

Keywords: coding, coding communities, formalisation, hybrid systems, programming

1 The terms of the question

A question is posed: assuming that one uses computers in the creation of art, is programming—writing code—a necessary or desirable skill for a creative artist?

A discussion of the terminology used will provide a fruitful starting point for an answer. To begin, the question implies that 'programming' and 'writing code' are synonymous terms. But programming can be read as the broader term. People even speak of 'programming' their video recorders. They give a machine a set of instructions to follow, which is indeed the essence of programming. Writing code, on the other hand, generally implies the exclusive use of text in the act of programming. In the former case, the 'programmer' interacts with an predefined algorithm by providing 'arguments' (numeric input) to its functions. In the latter case, they create the structure of the algorithm itself, putting both the functions and the arguments into place.

Both approaches typically involve a certain amount of abstraction—some awareness of structure—but the second normally involves a greater degree of abstraction and inevitably includes some degree of formalisation. Seen in that light, the question could be rephrased to read: what is the value of a formalised approach to the creation of art, as opposed to an intuitive one? This is a question that clearly predates and transcends the world of computers, and whose full scope exceeds that of the present article. Yet we may find some answers in our more focused, digital discussion, not least because any thorough discussion of the larger topic would lead inevitably to the issue

1462-6268/06/1701-0010$20.00

of blurred boundaries—the blurred boundaries between formalised and intuitive approaches to creativity, which in turn relates to the present idea of 'programming' as encompassing a range of activities.

2 Hybrid systems

Though there is some truth to the notion that those who approach their creative work in a more formalised way are more likely to be writing code, it would clearly be mistaken to state this in a categorical way. This is true partly because it is possible both to use prepackaged or graphic functions within a highly formalised context and, conversely (though perhaps less readily), to use code within an intuitive context. But it is also true because of the increasing number of hybrid systems available—i.e. systems that integrate graphic or parametric control with elements of 'coding'. *Max/MSP* is a perfect example of such a system (Figure 1). As with traditional coding, it enables one to create an algorithm 'from scratch', and in such a case requires that one be acutely aware of the flow of arguments to functions and of the orderly interchange of functions present. But it does this by graphically connecting one object to another, thereby suggesting a more 'user-friendly' form of programming—something that might be considered (rightly or wrongly) less sophisticated than 'coding'.

AC Toolbox is another good example of a hybrid system. ('AC' stands for algorithmic composition). The author, Paul Berg, has provided a basic graphic interface that uses terminology with which musicians are familiar, but in the parameter fields one could insert either a list of musically familiar values (see Figure 2) or fairly sophisticated bits of code. The program is implemented in *Lisp* (list programming) and bears some resemblance to it, but is less arcane and more musically intuitive. Like *Max*, it allows the user to find a place along a continuum of programming sophistication, including the ability to extend the program using the language in which it's written (*Lisp* for *AC Toolbox*, *C* for *Max*).

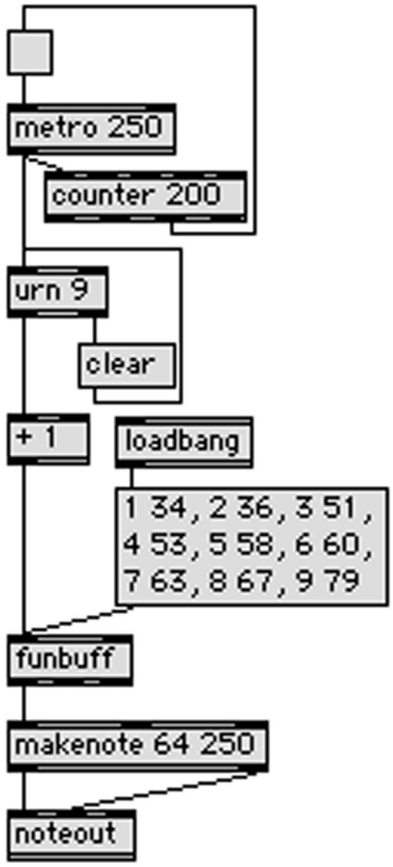

Figure 1. A *Max* patch that repeatedly chooses a random order for a group of nine notes from a minor pentatonic scale.

3 Choosing a method

For any given compositional task, there are often multiple programming pathways, beginning with methods that might not involve the computer at all and ranging to the use of pure code. The algorithms presented in Figures 1–3—all of which accomplish exactly the same task—reveal some of the advantages peculiar to each of the programs just mentioned.

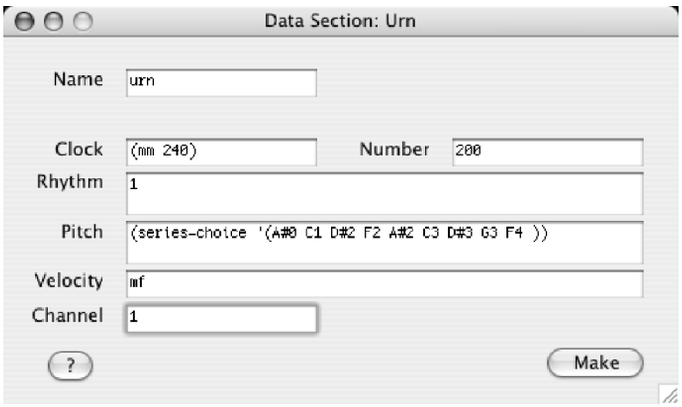

Figure 2. The same algorithm as used in Figure 1, now placed in an *AC Toolbox* GUI and substituting symbols in place of some of the numbers.

The *Max* version (Figure 1) might appeal to those who wish to see a graphical representation of the flow of input and output. Though more complex algorithms can become visually cluttered in this environment, this problem can be offset if, through a process of graphical organisation, one arrives at a greater conceptual organisation of the task at hand. More significant, however, is the fact that *Max* caters to interactivity, such that any parameter of sound can be altered during performance, making it possible to create a graphic interface customized for the peculiar needs of each composition.

In the graphic interface offered by *AC Toolbox* (Figure 2), the parametric organisation is unmistakably clear. In fact, one could say, in contrast to *Max*, that it is parametrically oriented rather than object oriented. And unlike *Max*, it liberates the artist from the need to consider the most basic levels of functionality and connectivity. On the other hand, it is not designed principally for interactivity and would therefore be a less likely choice for live performance.

The illustration in Figure 3 is also from *AC Toolbox*, but merely dispenses with the graphic interface. This requires the artist to expend more energy on learning the grammar of the language, but it is clearly a space saver and

```
(define urn
(make-data-section
250 200 1
(series-choice
'(34 36 51 53 58 60 63 67 79))
64 1))
```

Figure 3. Code written for *AC Toolbox* that accomplishes the same task as that in Figures 1 and 2.

in that sense is more efficient. If one wished to create multiple layers or complex algorithms, the text-only option has this advantage of enabling one to see large numbers of functions at once. It would also be a mistake to assume that this kind of environment is non-graphic, since the grammars and rules-of-thumb of any computer language result in graphic patterns that clearly help one organise the concepts represented.

In the end, the digital tool that one chooses involves several factors: the fitness of the tool for the task, the learning curve confronted by the artist, and the artist's personal proclivities. The overlap in the functionality of multiple programs (*Supercollider* is another that fits in the present discussion) may seem daunting for the programming novice who has to choose, but it is actually good news, since it not only provides a range of environments that suit individual tastes but also enables one to carry skills learned in one environment to another that may be needed for a new creative task.

4 Original impetus

Arriving at a place where one can choose a point along a continuum of programming options obviously requires a first step and a process of learning. For me, the first step was easy to take. When I first saw a demonstration of a rudimentary computer music program in 1985, I was immediately gripped by its capacity to provide me with immediate feedback and to give me sole control over the full cycle

of composition and performance. (This appeal had partly to do with my fears regarding live performers, but also with the fact that I was trained on the piano, an instrument given to solo activity.)

Perhaps I was fortunate that the early days of *MIDI* sequencing required that one work with alphanumeric lists: it was a good initiation in learning to see musical gestures in terms of discrete parameters and sets of instructions. Working with such lists, moreover, was not a difficult transition for someone accustomed to working with traditional music, for a musical score is really nothing more than a stored program expressed in a symbolic language; and musical composition, especially in the twentieth century, has often had much in common with mathematics.

5 Unintended preparation

Another step in the incremental process of learning to program occurred when I found myself for a time without any digital musical technology but still in possession of a computer. Partly out of curiosity, partly out of boredom, and partly from a desire to be among the knowing, I decided to learn a programming language. Fortunately, I chose one that is intriguing in its own right—*Forth*—so that my interest was sustained long enough to acquire some basic programming skills. (Part of the appeal of *Forth* lies in its resemblance to natural language—one programs in it by creating new words and placing them into a 'dictionary'—a resemblance it has in common with 'scripting' languages.) Later, I acquired more programming skills when I took a scripting course (in *HyperTalk*) that focused on producing interactive educational materials. In neither case was I involved in the composition of music, but I was unwittingly preparing myself for time when I would be.

6 A new partner

My experience with *Forth* was a short-lived one, partially because of the aforementioned lack of musical equipment, but also because I had no compelling task demanding its use. That compelling task arose later in the form of my dissertation, a composition for orchestra. It became clear to me during the early phase of its creation that I stood a good chance of getting mired in the making of endless minute decisions. And it became clear to me that the computer could provide me a way around that impasse. By having a clear sense of the trajectory of each parameter—a clear sense of the gestures and character of a section—I could instruct the computer to select the kinds of details that would lead to that outcome. I thus discovered algorithmic composition, and the computer became, as it were, a worker in my atelier.

My initial inclination as I embarked on this new journey was to write all my own routines using a programming or scripting language. This is perhaps the classic raison d'être for learning to code: the ability to fashion a tool that exactly suits one's purpose. Realizing, however, that this could lead me to drudgery that was merely different from the drudgery that I was attempting to escape, I sought and found a program that already had many routines written but which also provided broad flexibility and stylistic neutrality: the aforementioned *AC Toolbox*, a program that provides musicians with the flexibility and efficiency of coding but in a language appropriate to their discipline. (In this latter respect, its most obvious progenitor was *Csound* (see Figure 4)).

```
          Instr 1
kenv      linseg    0, 5, 10000, 5, 0
a1        oscil     kenv, 400, 1
          out a1
          endin
```

Figure 4. A basic bit of code from *Csound*, in which the arguments on the right are passed leftward to the functions, linseg (line segment), oscil (oscillator), and out (audio output). Though the order of the arguments is somewhat arbitrary, the functions at least have names familiar to electronic musicians.

7 Learning is composing

Nevertheless, even though the learning curve for *AC Toolbox* is not as steep as that for *Lisp*, it is not insubstantial. Before I could apply it to problems in my dissertation, I had to spend some time mastering it. But this provided an unexpected benefit: the composition of another piece. Learning to master the language engaged me in a process that was essentially compositional—namely, the conducting of numerous small trials in which one manipulates the parameters of music and uses the results to make corrections, and to conduct further, related experiments. This is a classic bottom-up approach to the creation of art, in which manipulation of the materials leads eventually to a sense of overall structure.

8 Composing is learning

Furthermore, since I was working with a new tool, I was engaging my discipline in a new way and exploring compositional thoughts that I might not have otherwise explored. In this case, I found myself working with musical parameters in an explicitly binary way—that is, considering the impact of using two contrasting states. For example, repeating a section and leaving all the parameters unchanged with the exception of one—changing, say, the rhythm from a state of extreme simplicity to one of complexity. The end result of this process was a piece for digital piano entitled *Delicate Outbursts*, from which I had now learned three important lessons: first, as already mentioned, the value of undertaking a new compositional modus operandi (in this case, writing code, or script); second, the potential of 'binary composition' (switching between two states in multiple parameters) for generating variations; and finally, that even though I was working with a machine and constructing mechanized, algorithmic processes, the end result could sound very human.

9 Quantisation is analysis

This last point is crucial. There are no doubt some who would avoid writing code on the basis that it is non-intuitive, non-gestural. Of course, it's certainly possible to use computers in a way that seems remote from intuition and gesture. But if one decides to implement gesture through the writing of code, the analysis that will be required for the quantisation and formalisation of gesture will result in a higher awareness of it and a greater degree of control over it. This kind of awareness helps me to avoid the mere repetition of inherited or learned gestures—to avoid what Iannis Xenakis called 'echolalia' (Xenakis 1992, p. ix).

10 The original hybrid

It was Xenakis who, in 1976, invented what was perhaps the first hybrid system of the type described above. In other words, one which included both graphic control and more sophisticated elements of programming, allowing for the use of a single system by a broad range of users. The system was known as *UPIC* (Unité Polyagogique Informatique) and included a large drawing tablet that could be used to synthesize sound graphically. Xenakis had children in mind as potential users when he designed the tablet, but the program also gave the more knowledgeable user extensive control over multiple parameters for every line that was drawn (Xenakis 1992, pp. 329–334).

Similarly, in both the systems mentioned above, there are many functions that can be accomplished either graphically or with text. In *AC Toolbox*, for example, a linear function could be implemented by simply drawing a line; alternately, one could use the *generate-line* function and accomplish the same task alphanumerically. Of course, neither of these programs is aimed at children, but they will accommodate more than one kind of user preference when it comes to the control of musical parameters.

11 A middle ground

In addition to providing a sort of middle ground between graphic and text-based systems, these programs also provide a sort of middle ground when it comes to their use of 'code'. In their degree of arcaneness, they are at the level of scripting, insofar as they present a simplified grammar and discipline-specific vocabulary, whilst still providing great versatility.

It was thus possible for me to easily implement a chaos function known as the Henon Map (see Figure 5) in my orchestra piece entitled *Analogies*. Given the fairly complex mathematics of this function, it is far less likely that I would have used it within the context of *C*, *Lisp*, or even a scripting language. The 'middle ground' nature of *AC Toolbox* freed me from unnecessary low-level programming but still gave me control over key parameters (see Figure 6), so that I was able to use it in a unique way and to make use of that characteristic of the phenomenon that I found most musically interesting (namely, the fact that it produces a pattern that is at once predictable and unpredictable). In addition, the discipline-specific nature of the program enabled me to export the results directly to a *MIDI* file and to subsequently import the *MIDI* file into a notation program, thereby shortening the distance between conception and realisation.

Figure 5. The Henon Map, at left, produced by the equation (xn+1, yn+1) = f (xn, yn) = (a - xn2 + b yn, xn). Interesting in the time domain because, though the overall contour is predictable, it is not possible to predict for any given iteration where on the map a value will fall.

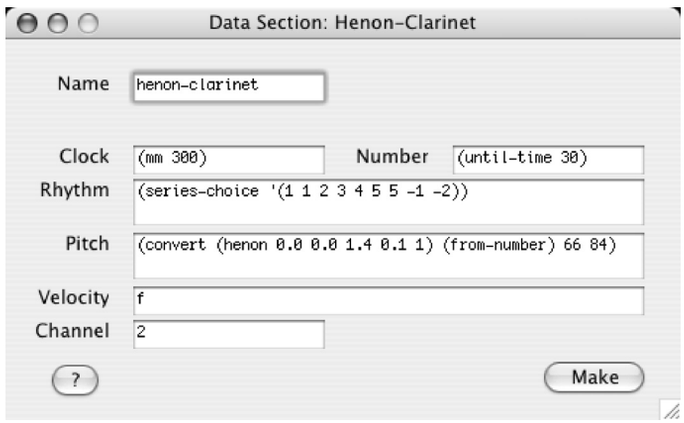

Figure 6. The Henon function mapped to the pitch parameter in a section from *Analogies*. The numbers 66 and 84 represent the pitches F#4 and C6, forming the low and high boundaries for the mapping. Other values control the degree of unpredictability.

12 Speed

Shortening the distance between conception and realisation is one compelling reason to learn some form of scripting or coding. Insofar as it enables algorithmic composition, it becomes possible to generate substantial amounts of coherently organized material in a relatively short period of time. I lived the proof of this in the summer of 2000, when I saw a listing for a composition contest in which composers were invited to compose music based on a painting by Andy Warhol. I was intrigued, but also hard pressed to finish another work, so I gave myself four days to compose a piece for the competition.

I chose the painting *Sixteen Jackies* and quickly derived a structure of sixteen movements each lasting sixteen seconds. In the painting, there were six different photographs of Jacqueline Onassis, each showing a different emotion. For each emotion, I drew a 'corresponding' curve in *AC Toolbox* and used the curves to control melodic contours (Figure 7).

I spent about two days generating the basic

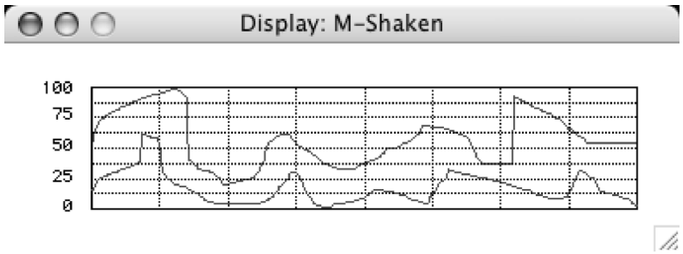

Figure 7. A free-hand drawing of the emotion 'shaken'. The upper and lower lines were used as boundaries to constrain pitch contours.

material in *AC Toolbox* and then another two refining it in *Finale* (a computer notation program), after which it was submitted and won third prize in the competition. I feel relatively confident in my assumption that I could not have accomplished the same task in the same amount of time without the programming abilities that I had acquired.

13 Coding communities

One of the benefits of learning to program with a discipline-specific programming tool like *Max* or *AC Toolbox* is the ability to connect with the communities that form around them. Large commercial programs and generic programming languages both tend to have large, diffuse user bases and authors that are remote from the user. By contrast, these programs that I have described as 'hybrid' and 'middle ground' often permit one to easily locate a centralised group of devotees and experts, as well as to interact with the author(s) of the program, thereby making it possible for the artist to shape not only their art but also the tool that they are using to make it.

During the five years that I have been using *AC Toolbox*, I have communicated with its author, Paul Berg, on many occasions. He has not only responded by fixing bugs but also by occasionally writing routines that cater to my creative imagination. For example, the original version of his program included a function called *ratio-choice*, which enables one to establish a set of weighted probabilities with respect to a list of musical events. I suggested that the function be modified so that the ratios could change gradually as the sequence unfolded, Shortly thereafter, he incorporated the suggested modification, and my subsequent delight and fascination led to the creation of a new composition, *Changing Weights*.

14 Conclusions

The artist's question of whether to code or not presents a somewhat false choice, for there appears to be an increasing number of programs (*Kyma* is yet another in the sonic realm) that present a range of options—options that both include, and lie between, coding and 'mere use'. They integrate graphic and text-based approaches and have the added advantage of using code that is discipline-specific and less arcane than generic programming languages like *C* or *Lisp*. In these kinds of environments, it becomes practical and fruitful to learn to code in an incremental fashion, since one need not move to a completely new environment to do so.

In addition to the most obvious and frequently stated advantage of coding—that it is infinitely customizable—we must add these:

1 the process of learning it engages one in a process that easily becomes a creative act;
2 it can easily lead one into a new creative modus operandi;
3 it forces one to analyse those elements that must be quantised and formalised;
4 if done with a discipline-specific program, it makes functions accessible for programming that would be difficult to implement in generic languages;
5 it makes it possible to accelerate the time between conception and realisation; and finally,
6 artists can become part of coding communities that both provide expertise and influence the design of the tool being used.

If more artists could see the world of coding as a continuum, whereon they could find a comfortable starting point and along which they could move freely, perhaps fewer would be intimidated by its arcane nature and instead exploit the advantages to be gained there.

References

AC Toolbox (2005) http://www.koncon.nl/ACToolbox, [software].

CSound (2005) http://www.csounds.com, [software].

Kyma (2005) http://www.symbolicsound.com, [software].

Max/MSP (2005) http://www.cycling74.com/products/maxmsp.html, [software].

Supercollider (2005) http://www.audiosynth.com and http://supercollider.sourceforge.net, [software].

Xenakis, I. (1992) *Formalized music: thought and mathematics in composition*, Pendagron Press, Stuyvesant, NY.

Ron Herrema is a composer, teacher and researcher working at De Montfort University's Music, Technology and Innovation Research Centre in Leicester, England. He is a native of Grand Rapids, Michigan and received his PhD in composition from Michigan State University in 2001. He composes both acoustic and electroacoustic music, specialising in algorithmic composition and in interdisciplinary approaches to music composition. Recent performances of his works have taken place in London, Singapore, Barcelona and the USA.