

Table of Contents

Software Art After Programming 1

Software Art After Programming

By Richard Wright

Ideas about computers and art have changed a lot over the past 40 years. Where once the possibility of making art with computers was treated with suspicion, today it is hard to find art that does not in some way reference the ubiquitous computerisation of society. Here, digital artist and theorist Richard Wright retraces the evolution of the debate and of the software tools that helped transform the computer-art relation.

The history of computing in arts practice is littered with the mental debris of its half-forgotten debates, unresolved problems and anxieties, and questions that have now become as obsolete as the Commodore 64s and VAX mainframes that accompanied them. Who can remember the art and technology projects of the sixties when the question of 'Can the computer make art?' allowed a generation of isolated computer artists to position themselves as a team of intrepid explorers setting out to cross a new continent without first waiting to find out whether it could support life. Under what conditions was the question ever first considered worthy of posing in the first place? Did the computer offer input into specific art issues, such as arts relation to other forms of scientific knowledge, to language, representation or the abandonment of the object? Or was it just intuitively realised that 'computer art' was at the forefront of a slow, inexorable computerisation of twentieth century society which would eventually demand access to every facet of human culture?

As computer hardware and the programming skills needed to operate it became more accessible, the question 'Can the computer make art?' was asked less and less often. By the beginning of the '80s artists were using the first personal computers to produce more varied kinds of work until, with all this activity growing, the question of whether art was possible on a computer lost all sense. There was a moment when the parameters of the question were redrawn, from 'Can the computer make art?' to 'Can a computer be an artist?', redirecting it into issues of simulated creativity and artificial intelligence. It was at this point that the first cracks of a coming schism in the community of computer artists became noticeable; this would go on to form the next stage in the debate. It seemed to a growing number of artists that as the complexity of software increased, so many new possibilities for the human artist were appearing that the prospect of deferring to a machine artist seemed almost indicative of a lack of imagination.

Although the computer seemed to have made its case as a machine of creative potential, there now emerged the question of how to efficiently leverage all this creativity. By the late eighties, the interactive interfaces and simplified menu commands of personal desktop systems that had helped to cause this ground swell of activity had firmly refocused questions on the artists themselves. Were the pre-packaged functions, options and parameters of the new art applications sufficient to cover all artistic fields of inquiry, all aesthetic nuances, all personal idioms? Or would it always be necessary to have recourse to the precision and particularities of programming languages in order to ensure that no desire was left uncatered for? 'Do artists need to program?' became the burning question at SIGGRAPH panel sessions and electronic art festivals.

To some extent this divergence between programmers and program users masked the fact that they had become two sides of the same coin. As the argument went, the artist-programmer would regard '...software not as a functional tool on which the "real" artwork is based, but software as the material of artistic creation', as the Transmediale Software Art jury statement would phrase it much later in 2002. On the other hand, for program users, programming was only ever a means to an end. Yet it was their fixation on this end that hastened their acquiescence to the means of their programs and the reconfiguration of their practice by programmers. 'Is the computer a medium or a tool?' Yes, it was

true that some artists were only interested in software 'tools' that were totally subservient to their subjectivity, but it was a subjectivity that was now mapped onto minutely variable parameter lists and option check boxes, mirroring the remoteness of the artist's precious and peculiar visions by burying its origins deep within the recesses of multiple menu layers. Aided by the runaway success of packages like Amiga's Deluxe Paint, Adobe Illustrator and Photoshop, software manufacturers were redefining the creative process as a decision making process converging towards a predetermined ideal goal.

The problem was also attacked from the opposite direction by a top-down system design employing pre-sets, wizards, helpers, macros and plug-ins that pre-empted the creative process by offering a one button solution to achieve those essential lens flares, ripples, rollovers and drop shadow effects. The users of programs now found themselves programmed by their very own favourite artistic effects, expressed as a suite of easy to use software extensions. In the end, both artist programmers and artist program users produced artwork that was about the software that had produced it. Both became caught up in a wider move to rewrite society in terms of information processing.

By the early '80s the artist Harold Cohen had developed software to automate his own personal artistic style. A former successful gallery painter, Cohen still works on a suite of artificial intelligence programs called AARON that seek to encode his earlier painting practice. Cohen had always insisted that the content of his work was the software itself, and always exhibited the entire process in the form of a live computer connected up to a mobile painting device or 'turtle' that would scuttle over his canvases. As he told his students, 'Don't ask what you can do with the software, ask what the software can do.' But Cohen's work now seems to function more as evidence of a historical transition that occurred over his working life and reached its culmination during the '90s. While we have been watching Cohen's computer prove it can recreate art, other computers have been recreating our whole society in their own image. But this new image is not the image of the expressive subject that is simulated in Cohen's work. It is the image of the subject as a node, a switching station for providing feedback to calibrate the central processing system, the individual's expressive utterances only called upon to ensure their movements are correctly synchronised. The artist programmer of today exists in relation to a whole culture that has the computer as its central organising technology. The pervasive quality of software culture and the resultant normalisation of computer use have made it impossible to maintain the conceptual categories that underpinned previous debates. In a world where artists use software to write software that will be seen by virtue of other software, questions about the 'aesthetics of the code' become a symptom of not being able to see the wood for the trees. Programming is not only the material of artistic creation, it is the context of artistic creation. Programming has become software.

One interesting example of the end game of the debate on 'Computer Art' is a piece of artist's software called Auto-Illustrator. Written by Adrian Ward around the year 2000, Auto-Illustrator was the prize winner of the first competition for Software Art organised by Berlin's Transmediale media art festival in 2001. Ward describes the work as a parody of commercial art and design packages like Adobe Illustrator, specifically of their pretensions to provide functionality and user control. In contrast, Ward fills his package with 'generative art' tools that explicitly try to automate the drawing process. The appearance of Auto-Illustrator when running is much like a typical menu driven art and design package with the exception that the tool palette and effects filters incorporate generative algorithms. For instance, the Pencil tool adds wiggles or sweeps to your strokes, while the Oval tool will use settings like 'childish' or 'adult' to control a sprinkling of little faces. Some tools like Brush seem entirely random in operation, while some filters like 'Instant Mute Design' will reproduce an entire iconography designed to appeal to the Digerati generation.

In fact, many of these generative techniques are strikingly reminiscent of various experiments in computer art from over the last thirty years. The line tools generate scribbles using algorithms almost certainly related to the stochastic perturbations of Frieder Nake or Peter Beyls while the 'bug' tool roves around the screen using the same principles as Harold Cohen's turtle graphics engine. Even the icons of the 'Instant Mute Design' effect are almost identical to Edward Zajec's permutations of cubic modules. In this way, Auto-Illustrator is like a compendium of classic computer art programs but now presented as a list of menu options with conveniently editable parameters. Presented in this context, the individual aesthetics of each of these venerable pioneering practices are erased, leaving us with more of a confusion of idiosyncratic styles. From this viewpoint, Auto-Illustrator's 'generative tools' actually pastiche the chaotic 'feature mountain' of bloated modern software systems, as they are commonly disorganised by the superabundance of toolbars, drop-down lists and floating inspectors. Instead of defining a drawing function, it might have been more relevant if Ward had made his 'bug' tunnel into the dizzying depths of cascading sub-menus and option boxes to find that single cherished function with which the user nurtures their unique style. Ward actually states that wider issues such as interface design are of no interest to him and describes 'consumer-based application software' as his chosen medium. Auto-Illustrator is successful in its intention to parody the functionality-as-expression of mainstream software design, but only at the level of coding. By not addressing the wider user experience it is unable to think outside of the window box in which this functionality is now defined.

Since Auto-Illustrator's release there has been at least one attempt to account for a contemporary digital aesthetic with reference to the design of a family of software packages and related technologies. In 2002 the theorist Lev Manovich published 'Generation Flash', an essay in which he tried to characterise a then prevalent cultural sensibility. Manovich referred to the prevailing visual style of Flash, Shockwave and Java based multimedia as 'soft modernism', a reaction against the clutter of postmodern eclecticism that returns to an elemental 'rationality of software'. Aesthetic motifs are defined by Manovich in terms of technologically motivated processes: instead of appropriation we simply have the data sample, a basic operation in the new mode of cultural production. Another cultural building block is the network, and therefore also one of the terms of a new critical language. These operations (networking, sampling) are applied in new modes of expression like data visualisation. This can be seen, for instance, in Futurefarmer's They Rule project in which the directors of the USA's top corporations are cross referenced to purportedly reveal a web-like pattern of interrelated allegiances. For Manovich this kind of work replaces older forms of authored representation by giving us the tools to objectively analyse raw data and deduce the necessary conclusions.

Although Manovich's detailed analysis of the structural basis of new media adds an absolutely essential dimension to new critical tools, the approach risks being interpreted as a form of technological determinism once we lose sight of a specifically cultural perspective. For example, our understanding of the workings of the corporate world order do not arise automatically out of its most common data visualisations, such as the stock market fluctuations diagrammatically portrayed on the Financial Times website. Not all visualisations are equal. At one point Manovich argues that the 'neo-minimalism' of the Flash style arises quite naturally from the practice of programming – the pixel thin grid lines, restricted colour palettes, abstracted symbols 'ALWAYS happens when people begin to generate graphics through programming and discover that they can use simple equations, etc' (Manovich's emphasis). This is indeed the case where programming is taught within a certain computer science tradition, but it is now impossible to discount the influence of scripting environments such as Flash. Not all programming practices are equal.

Other discussions of Flash have merely tended to shift the technological focus, such as whether the limited bandwidth of the web was the most significant reason for the linear aesthetic of vector graphics. At other times it moved on to question the 'openness' of the Flash graphics standard,

whether Macromedia would ultimately allow programmers to leverage the full potential of its functionality. However, the ‘functionality’, ‘rationality’ or ‘potential’ of software will always be strictly unknown. It is the ‘user experience’ of software, how its operations can be made to appear rational, how it can accommodate a functionality where none may preexist, how easily a technical ‘potential’ can be perceived and engaged with that should form the basis of software critique. It is possible to trace many formative influences on the Flash style not to the code itself, but to the conditions in which it is written. Programming is now often practised in the form of ‘scripting’ languages that are integrated into mainstream art and design software applications. This makes artist programmers and program users both subject to the same philosophies of system design that hold sway in point-and-click style desktop packages. By examining these environments we can find many ways in which they funnelled Flash Actionscript or Director Lingo programming practice into nourishing certain wider cultural sensibilities during this period.

Multimedia scripting languages like Flash Actionscript tend to differ from conventional programming languages by offering access to a library of functions that are specific to that particular multimedia application. This easy access to a set of predefined ‘events’ such as mouse clicks, drag actions and rollovers is somewhat analogous to the way a software user’s practice is structured in terms of the predefined configuration of menu commands, option boxes and plug-in effects. These library functions that populate the programmers imagination with a readymade vocabulary of discrete interactive ‘behaviours’ can be attached to individual multimedia objects – button triggers, sprite actions, sound effects, linkages, etc. In what came to be known later as ‘Flash 5’ style scripting, Actionscript behaviours were scattered throughout a project, attached to various buttons, icons, movie frames or nested away in other clips, producing a particular fragmentation of the programmers flow. Actionscript therefore tended to differ from typical program development environments by identifying code with graphical and other concrete entities that would become principle actors in the interactive scenario. When combined with the instancing abilities of the Object Orientated Programming philosophy, Actionscript became very efficient at applying these code segments to multiple copies of ‘semi-automated’ graphic elements, sprites, movie clips and sounds. As implemented in multimedia authoring software like Flash, Object Orientated Programming actually fostered an ‘object orientated’ approach to interactive art and animation.

The point here is to look at Flash at the moment at which its patterns of techniques and processes re-emerge as motifs that can enter consciousness and practice on an aesthetic level. To start with we have an authoring system that orientated the user towards the replication (or ‘birthing’) of multitudes of objects and orchestrating complex yet concise interactions between them. It is even possible to identify the most common form of mathematical expression that was used to regulate this interaction during the millennial Flash period. There is a single line of code that appears over and over again, a simplified expression that produces a distinctive dampening effect on a moving object before it finally comes to rest. It was easy for Flash users to apply this expression to any or all of ones objects and events until it produced the classic Flash ‘wobble’. A Flash site became a constellation of rippling, bobbing, trembling buttons, icons, eyeballs, legs and rollover items as if someone had poured a bucket of water into your computer monitor. In the open source spirit, the Flash community ensured that such expressions were quickly disseminated until they became an almost universal kinetic attribute.

The Flash style was integrated, via its web browser plug-in, to other desktop based work and leisure patterns of activity. By keying into the internet gold rush fever, Flash art was turned into a highly visible design component of the dotcom boom era. This new informal space imbued Flash art with the role of a distraction, a demo or toy, making any more demanding appreciation of its fluid stylistic and tactile qualities unnecessary. The net culture of the time also provided a preexisting discourse in which it’s visual aesthetic could be interpreted and flourish. Echoing the ubiquitous net-cultural meme of the ‘digital Gaia’ – an ecological interpretation of the web of globally interconnected and independent

agents – foremost Flash designer Joshua Davis commented: ‘...our work should reflect the nature of a fern and be comprised of tiny little objects that all talk to each other. The more we add these little objects, the more complex and intense the nature of our work becomes.’

There are many more factors that could be marshalled to ‘explain’ the Flash style. But as far as practising artists are concerned, how can we get a handle on such a deluge of widely different factors, some of which seek to align us with a particular model of subjectivity and others which just seem like arbitrary collections of protocols? How can we forge a path through layer after layer of designed information to form ways of working not pre-empted by the predicates of current software culture?

There are some emerging ideas that might help. One of these is the ‘techno-aesthetic’ – different motifs that permeate these technological, social and cultural levels. The emphasis here is on how specifically cultural forces can form technology into a means of expression that is able to exceed its most obvious properties and structures. One software art example of this in action is Mongrel’s often-cited Linker project of 1999. Developed to support a series of story telling workshops for the non-expert computer user, the software is a highly stripped down system that simply allows users to load and make connections between a collection of digital elements – images, text, video, sounds. For a start, this transfers an emphasis on the practice of the software to the practice of the user. Compared to the other examples, Linker coheres around a figure that unites its levels of thought and construction yet retains an open space in which imagination can breathe. As theorist Matthew Fuller described Linker, ‘It relies on the simple function of doing exactly what the name says it does – link things. Here, the poetics of connection forms a techno-aesthetic and existential a priori to the construction of a piece of software.’ This aesthetic is made explicit when the software is first launched – it displays a map image of its three by three grid of interconnected regions. Linker is constructed around this image of itself that communicates and instantiates its underlying algorithmic structure, creative use and conceptual model. It is this figuration of itself as an idea that makes Linker art as well as software.

The debate about Linker was unfortunately always limited to its mode of production and the social constituency of its intended user group as though it had been designed as a tool of social engineering, ready to arise fully formed out of a sub-menu check-box list of community ‘needs’. But discussions of DIY empowerment, Open Source and the ‘sociability’ of software are presumptuous without any attention to the context in which imaginative ideas can grow. When we look at the kinds of applications that have actually resulted from Linux we simply see copies of standard Microsoft functionality. The Open Source model of production is a dead end without an equivalent ‘model of creativity’, defaulting instead to a wannabe culture. Instead we should look for inspiration in practices that could nourish a poetics of data ‘copyability’ such as plagiarism and detournement, as noted by writer Josephine Berry. But unfortunately free software developers do not prioritise this aesthetic context which is what has the power to determine whether software will enable or restrain its user’s perceptions and mode of action.

It is not a matter of the different technical abilities of software or of how much it costs, but of how easily a technical potential can be perceived by the user in a way that motivates engagement. When software is written, choices must be made about which data fields carry value, how the display of information forms contours of meaning, how the modelling of the interface moulds the subjectivity of the user. The question of whether artists should learn to program is replaced by the question of what kind of programming. Which programming practice has the most ‘open aesthetic’, capable of envisaging software that is not just the product of an arbitrary confluence of techniques or a slavish mimicry but is aware of all its possible formative cultural and philosophical categories and values.

For the first generation of artist programmers there was hardly any information society in existence, certainly not one within reach. In the early eighties during a period when the launch of the personal computer marked a radical shift in computer culture, artist Harold Cohen stressed the importance of asking the right questions. Now that we live in a world in which his AARON program is downloadable as a screen saver it is time for us to extend his question – ‘Don’t ask what the software can do, ask what it can do to other software.’

Auto-Illustrator: www.auto-illustrator.com

Joshua Davis: www.joshuadavis.com

Linker & 9: www.linker.org.uk, www.9.waag.org

AARON screensaver: www.kurzweilcyberart.com

Acknowledgement

This article was based on research supported by a grant from the Arts and Humanities Research Board

Richard Wright’s <richard AT dig-lgu.demon.co.uk> new film *Foreplay* has been described as ‘a porn film without the sex’. He is currently working on ‘Catastrophic Code’, a large software project to create a 17th century operating system