Programmed Graphics in Computer Art and Animation

Dr. Mike King

Visual Outcomes in Computer Graphics

Algorithmic vs. Arbitrary: In looking at the work of artists, designers, and animators using computer graphics systems I have found it useful to invent the terms arbitrary and algorithmic synthesis from primitives. (I have introduced these terms in previous papers, [1] and [2].) In any computer graphics system one is synthesising an image from primitives, which may range from the pixel to 2D geometrical shapes to 3D solid forms. In arbitrary synthesis from primitives the system has no control over the placing and attributes of each primitive: the artist is following their own intuition and reasoning, to which the computer is not party. In algorithmic synthesis from primitives the artist or designer yields up control to some degree by imparting to the computer a set of rules (an algorithm) by which it should incorporate or manipulate successive primitives, which may include a random element. The artist will initiate the starting conditions, and allow the computer to take over.

Most computer-generated or computer-manipulated images will be the result of a balance between the two approaches of arbitrary and algorithmic synthesis, as defined above, but it is my strong belief that the computer offers something radically new to the artist when they explore the algorithmic side of image generation. As well as a range of imagery not realisable through other methods there is also the attraction of serendipity: there is the possibility of an unpredictable but satisfying outcome. The photographer knows this feeling for example, as he or she watches an image develop in the dark-room; as does the bronze-caster when the mould is broken and the new-born surface is revealed. In computer graphics an image may slowly develop, or only be made visible after an over-night ‘batch’ render; even with the promise of ever more powerful systems it seems that there will always be some waiting involved, giving rise to a somewhat addictive sense of anticipation. The anticipation arises in part from an aspect of programmed graphics (or any non real-time graphics) which is that the outcomes are deterministic but to some degree unpredictable. They are deterministic because the same starting conditions produce the same end-result (unless we input truly random data, for example from a Geiger counter), but unpredictable in varying degrees because of the complexity of the process.

Existing Software: Most off-the-shelf software packages offer some algorithmic features, but to really exploit the potential of this approach it is usually necessary to program the computer to a greater or lesser extent. It may however be helpful to consider the types of software available to the artist before looking at the necessity of programming. In my PhD thesis [3] I outlined a simple taxonomy of computer graphics software, and elaborated on it in some later papers. I list it here with a few changes to bring it up to date:

Paint bit-mapped graphics (paint systems)
Draw 2D vector (object-oriented) graphics (draw systems)
3D including modelling and rendering
Page layout formatting of text and graphics (bit-mapped and vector)
Animation time-based versions of paint, draw or 3D systems.

The distinctions made between the first three categories are the most important: software may attempt to integrate two or more of these categories, but in practice this is rare. The three types of package: paint, draw and 3D, are used for quite different visual outcomes, and are quite different to use. Page layout is not directly the concern of this paper, but is included for completeness; it is as well-defined an entity as the previous three. In terms of animation systems we see examples of time-based paint or 3D packages, but there are no widely-available draw or vector animation systems for PCs or Macs, the existing ones being high-end systems for the cartoon industry.

Paint systems, draw systems and 3D are used for different visual outcomes, though each one may be 'bent' towards the imagery more normally associated with one of the others. Paint systems are good for photographic treatment, and for painterly effects; generally for subtle effects. A draw package is good for diagrammatic imagery and more technical illustration work: it is resolution-independent, so output quality can be very high, but the visual outcomes usually lack the expressivity or subtlety of paint-system work. 3D packages are used to simulate 'reality' through the processes of virtual sculpture (or virtual fabrication) and virtual photography. Using these systems we see mainly the visual results of arbitrary synthesis from primitives: the results are highly dependent on the artist or designer. The algorithmic synthesis provided by these systems are limited to the provision of geometrical shapes, and variants on 'inbetweening' – routines that allow one shape to change into another over time or space or both.

User-Written Software: If an artist, designer, or animator is writing a program for generating an image (rather than writing a complete software package) the resulting program does not easily lie in any of the previous software categories. However, the data generated by the program will fit into one or other category, for example a piece of code may generate a bit-map (for a paint-package), a vector file such as PostScript (for a draw package), or 3D data for a 3D modelling package. The user-written software will generally be written to exploit an algorithm, and hence will involve algorithmic synthesis from primitives. It is possible however to write a purely arbitrary 'shopping list' of primitives, but in most cases it would be easier to use an existing software package to do so. (One of the early programming exercises we give our students is to do just that: a piece of 'C' code to draw a figure – the resultant code looks similar to the PostScript output of a draw package used to generate the same drawing.)

With programmed graphics, where algorithmic synthesis from primitives prevails, it is easier to classify the visual outcomes than with software packages. Many algorithms are based on simple mathematics, or more often geometry, in the sense of geometry being that branch of mathematics with a visual basis. Herbert Franke has this to say about mathematically based images [4]:

‘If one is engaged, as we are, with an inventory of all mathematical branches and with an interest in visualising all forms that come to light, one can obtain plenty of forms, shapes and structures never seen before – an expansion of our treasury of forms. Many of these forms have considerable aesthetic charm. According to the usual criteria we cannot call them original works of art. But they can be considered elements available for new creations and can be used to develop artworks.’

In this quote we have a basis for looking at visual outcomes from programming: a mathematical taxonomy. I have explored this idea to some extent in [1a] and [2a], making a binary distinction between ‘classical’ and ‘fractal’ geometries, the latter having developed only in the last century, and their full importance only made visible through the use of the computer in this century. Briefly the taxonomy I have proposed looks like this:

Classical Geometries
¨ geometrical primitives such as circles and rectangles;
¨ mathematical functions such as roots, squares, exponential curves, parametric curves and surfaces;
¨ Lissajou’s and related figures, involving sine and cosine calculations, including waves;
¨ Patterns, including nets, bands and tessellations.
Recursive Geometries
¨ Iterative functions (recurrence relations);
¨ random numbers (these are usually based on iterative functions);
¨ fractals and graftals;
¨ particle systems;
¨ growth models;
¨ linear and array grammars;
¨ Markov chains.

A great number of artists have used variations of sines and cosines to generate imagery of waves or to manipulate 3D solids and textures. Many software packages are providing variations on waves or ripples in 2D, 3D, and also for texture maps. Patterns, often classified under linear repeats (friezes), area repeats, and tessellations (interlocking outlines) are another very common use of programming techniques. I discuss these further in [1b], and also describe some of their computer implementations in a system called ICAS in [2b].

Image Processing: Image processing differs from the previous methods of algorithmic synthesis from primitives, in that it does not involve the continued addition of primitives under algorithmic control, but the re-computing of pixel values in an existing bit-map (for example in a scanned photograph). Image processing involves such a wide range of techniques that one can fruitfully sub-divide them into categories:

Point processes – each dot or pixel is processed regardless of its neighbours
Area processes – neighbouring pixels are taken into account
Frame processes – more than one image combined
Geometric processes – pixels are moved from their original positions.

I have taken these categories from [5]. PhotoShop, a typical image-processing programme for the Mac and Windows on the PC, offers a wide range of these processes, covering all four categories. The geometries involved in the last category are often simple displacements of the image based on waves or spirals, but in principle any of the algorithms in the taxonomy outlined above could be used. Graham Harwood [6], whose work is discussed below, has extended the point process with the replacement of pixels by line segments, whose attributes depend on the pixel intensity (see fig. 1).

Artists using Programming

The first computer artists could be said to be people like Franke and Ben Laposky, who experimented with images generated with oscilloscopes as far back as 1950. Laposky called these early experiments 'oscillons' or 'electronic abstractions' according to Franke [7]; they were generated initially with analogue computers. These techniques readily translated into digital computer graphics, and have become part of the visual language of computer imagery ever since. Similar mathematical techniques were exploited by John Whitney Sr. in his pioneering animations of the sixties [8]. Whereas Whitney's animations were abstract, and based on the 'classical' geometries described earlier, another pioneer, David Em, worked with 3D imagery based originally on programs written to simulate planetary bodies [9]. By the time that he entered computer graphics, in 1975, simple paint systems were available, and his work is a mixture of paint system and programmed techniques. Much of the programming was done by James Blinn at the Jet Propulsion Laboratory, and Em was able to use and modify pioneering code that give 3D surfaces rich textures.

Whitney and Em are examples of a tradition of artists working in large institutions (IBM and the Jet Propulsion Laboratory respectively) whose aims and environment are far removed from that of the painter. William Latham, whose work I discuss in more detail below, is a print-maker and sculptor who is a modern exponent of that tradition at IBM [10]. There are other examples of artists working collaboratively with programmers: John Pearson employed under-graduate programmers to produce a series of combinations of arc segments used in his paintings [11] and Harold Cohen, discussed below, works with artificial intelligence experts [12]. Other artists have worked largely outside of institutions and programming help, using their own machines and skills: Mark Wilson [13] and Edward Zajek [14] are examples.

The first edition of Franke's 'Computer Graphics – Computer Art' [7a] gives an introduction to work of the programming pioneers, and also presents an early discussion of the aesthetic of 'Computer Art'. In 'Digital Visions' [15], a more recent publication that includes imagery from software packages, Cynthia Goodman gives a good non-technical coverage of the early computer artists: the bulk of their work was realised through programming because nothing else was available until the early paint and 3D packages came out.

William Latham, as mentioned above, has followed in the footsteps of pioneers like Whitney, Em, and Cohen, by collaborating with a large institution, in this case IBM. He is presently full-time Research Fellow at the UK Scientific Centre in Winchester UK, and works in close partnership with Stephen Todd and a team of programmers and researchers. He has developed a unique 3D computer graphics system that allows him to mutate 3D forms and animate them. At the heart of the creative process is a system component called 'Mutator' which allows the generation of basic forms, their evolution through random mutations of gene 'vectors', and their breeding and selection, giving Latham the opportunity to act as 'creative gardener'.

The 3D models that Latham originally works with are the result of combining fairly traditional 3D computer graphics primitives (cube, sphere, cone and so on), using Boolean modelling controlled through a scripting language called ESME. The ESME programs are created with a text-editor, and give algorithmic control over the placement of primitives, including the use of a programming technique called recursion, which in this context corresponds to the recursive geometries mentioned earlier. Fig. 2 illustrates an ESME code fragment. Mutator has now taken over this 'low-level' text-driven algorithmic control by the use of the metaphors of evolution and breeding, as in fig. 3, which shows evolving forms and the main Mutator menu. This is an interesting example of algorithmic synthesis from primitives controlled at one level by writing code, and at another level by an interactive user interface. Fig. 4 illustrates a 'still' image as a final piece generated by the system, and chosen by Latham for exhibition in the form of a large photographic print. The illustrations come from the book 'Evolutionary Art and Computers', by Todd and Latham [10a], which describes the systems in detail. The question posed by Latham's use of evolutionary algorithms is – what takes the place of natural selection? His answer is not any form of 'survival of the fittest' algorithm, but in his approach of a flower gardener – he simply selects according to his own visual sensibilities.

Karl Sims, a computer artist with a science background, has used a similar approach to Latham in his 'Panspermia' series, creating scenes with 3D plants controlled by genetic parameters [16]. These are 'bred' together, using a 'survival of the prettiest' to select the final set of parameters – the results are quite different to Latham's work however, tending to a kind of realism involving plausible vegetation. Note that Sim's selection criteria are similar. Sims is artist in residence at Thinking Machines Corporation – the computing power needed for his work, as with Latham's, is probably too expensive at present for an artist working alone.

Harold Cohen, a successful British painter in the mid-'60s, wrote his first computer program in Fortran in 1968, at the age of 40. This lead to the evolution of a single artificial-intelligence program called 'Aaron' which became capable of generating figurative drawings by the mid '80s. Cohen's work is pioneering, unique, and important, and illustrates one major attraction of learning to program: you can achieve something unique, pioneering and important. This is meant quite seriously, although his work is often overlooked, despite one-man (or should it be one-man and one-computer) shows around the world, including at the Tate Gallery in London, England [17]. Cohen has used programming to incorporate rules of painting or composition, derived from an analysis of his own work and others, including children's drawings. The imagery is not based on maths, geometry, or algorithms in the sense of previously described work, but on decisions, or at a more basic level, the if-then-else statement. The work does however fall into the category of algorithmic synthesis from primitives, where the algorithms are not so much derived from maths or geometry, as from what Cohen terms cognitive primitives. In Pamela McCorduck's book 'Aaron's Code' [12a] Cohen says: "The program [Aaron] has always been structured in terms of cognitive behaviour, not in terms of form." (This is in contrast to Latham's work, which is based on form – 'evolution of form'.) The cognitive primitives in the early versions of Aaron relate to the subdivisions of a surface in a series of entirely abstract works entitled "Three Behaviours for Partitioning Space", leading much later to the representations of rocks, plants and people in the explicitly figurative "Eden" series. Cohen says of the direction of his work:

"Aaron ... does not do naturalistic pastel drawings, not because it would be difficult, but because it would be inappropriate. But I would also not have it in mind to draw those interminable geometrical figures popularly identified by now as 'computer art'."

According to Pamela McCorduck, Aaron was written in the C language up to 1990, after which it was translated, or re-written, in Lisp, a language more commonly associated with artificial intelligence. Cohen has benefited from collaboration with DEC, with the loan of DEC VAX machines for his shows, and eventually an outright gift of a machine from the corporation. As with Latham's and Sims' work, a lot of computing power is needed for Cohen's work – in 1979 his machine had 64K RAM, while his present machine has 34 MB, a factor of about 500 which probably reflects the overall increase in power available to him over the period.

Graham Harwood, a colleague at London Guildhall University, is founder of 'Working Press' and originator of many counter-culture publications in the UK [6a]. He began to use computers as a direct result of his use of photocopiers, and to achieve some of the aims set out in the 'Festival of Plagiarism' [18]. In 1987/88 there was little software available for affordable computers that could do the kind of image manipulation that he was interested in so he wrote his own software in 'C' on a PC compatible. Although he says now that he could probably manage without his own code, he still intends to continue programming for specialised image-processing effects, in particular effects that could give the sense of appropriation, corruption and degradation, as shown in fig. 1. Graham rarely uses maths or geometry as a starting point, preferring to start with a visual effect and work backwards to the required code using common sense. Although he makes little use of the algorithms developed by computer graphics researchers and published in places like the SIGGRAPH conference proceedings, he does like to feel part of a programming community, partly as a counter to the traditional 'precious' status of art and artist.

For the future Graham is looking at the programming of plug-ins for existing packages like PhotoShop, and also to writing interactive works where the user can make their own way through one of his stories. His preferred medium is the broadsheet or booklet or magazine, on the basis that there are currently not enough potential readers equipped with the multi-media technology for interactive works. As that changes however he envisages distributing his work on CD-ROM.

Richard Wright, also a colleague at London Guildhall University, is known for his computer animations 'Superanimism' [19] and 'Corpus' (see fig. 5), the latter being jointly produced at the NYIT and London Guildhall University, and for his writings on computers in culture. Richard was Research Fellow at IBM UK before Latham took the post, and developed his interests in computer graphics as a visualisation tool in science [20], writing a ray-tracer and other software that has been used both in his animations and in the renderer for my 'Sculptor' images [21]. His original interest in computer graphics programming was as an alternative to manual methods of production, but also for the new imagery available through mathematical techniques. Although much of the software that he has written is now commercially available, he still finds many techniques only realisable through programming. Richard also makes the point that familiarity with computer graphics programming journals means that he is much more aware of leading-edge techniques that take many years to reach commercially available packages, and which he can take advantage of. Another interesting point that he makes in favour of programming is that it tends to allow the artist to impose their 'mark' in a more consistent way than when using a range of commercial packages. (I have also found this to be the case with the 'Sculptor' series, where attempts to continue the work in commercial 3D systems has lost a certain individuality.) His current project 'Heliocentrum' has attracted an Arts Council of Great Britain award, and involves use of an advanced liquids visualisation technique. He has used fluid dynamics software adapted from the SIGGRAPH paper by Cass and Miller [22], illustrating the previous point regarding leading-edge techniques.

Writing Programs

Having looked at programmed graphics in general, and some case studies in detail, we can better ask the question: how has the rationale for programming for the artist changed over the last forty years, and in particular just recently with the availability of good art software running on affordable hardware? What other factors are there?

A feature of much early computer graphics was its vector orientation: output devices such as screens and plotters drew lines, and many algorithms made use of this. With the replacement of the vector technology with raster technology, the capacity for continuous-tone imagery arrived, and also, more recently, the appropriate software packages. Raster graphics has probably made writing software more difficult, firstly because of so few standards concerning graphics displays and colour output (in the old days you had lines and that was it – if you wanted you could change the pen!), and secondly because one is drawn into much more ambitious projects. This problem – of more ambitious projects – leads also to the need for more powerful equipment, as we see in the cases of Latham, Cohen and Sims. More powerful computers can leave one with a smaller range of commercial art software, but this paradox is easily understood if one considers that few artists and designers are likely to own top-end Unix boxes or supercomputers, so there is a very small market for good art and design packages for this type of hardware.

Historically most programming ventures by computer artists have taken this form: the writing of small stand-alone programs for certain visual outcomes – Cohen's Aaron is a rare exception in being a single program that grew over a period of nearly 25 years. The early programmers would have used Fortran or Algol, and possibly machine languages to interface with plotters and so on. Nowadays the more likely languages are C and Basic, with compilers and interpreters readily available for Macs, PCs and Unix platforms. In each case a library of basic graphics commands is included within the compiler, but with no commonly agreed standards: the chances of a command to draw even a filled rectangle the same way on the three types of system are small. However, with such a range of commercial software packages now available on the Mac and PC, each offering a huge functionality, some of the reasons for writing programs have gone. Some of the reasons are still with us however, and many software developers are aware that their package, however huge it grows, will never satisfy the needs of all their customers. Hence the introduction of programming 'hooks' into the system, allowing a third party, or even the customer, to write specialised bits of code.

An example of software with programming 'hooks' for the user is Ani Pro, a 2D animation package running on PCs, at present restricted to 8-bit colour (256 colours), and produced by AutoDesk. It has a programming interface for the user consisting of a C-language interpreter (called Poco) built into the package, which the user accesses from one of the menus. The programmer has access to a well-documented library of functions that represent the systems interface, drawing tools, inks, and time-based commands. Poco seems best suited to interaction programming, as its execution of code is approximately ten times as slow as compiled code; however, the code can be compiled to run faster with specialist compilers. 3D Studio is a professional-level 3D animation package also produced by AutoDesk, and uses C-language plug-ins, called external processes. Unlike Ani Pro these are externally written and compiled code generated by a protected-mode compiler (a protected-mode compiler produces code that makes use of the more powerful PCs, in particular the large amounts of memory needed). The plug-ins are divide into four types:

Image processing
Procedural modelling (e.g. waves and spirals)
Procedural animation (e.g. particle systems)
Procedural textures.

These plug-ins allow the incorporation of algorithmic synthesis of primitives into an interactive package, an ideal opportunity for the artist/programmer to extend a package.

The computer artist may start by writing some specialised pieces of code, and find that it may be worth developing it into a package. The end-result may be just for their own use, or may find a wider use, either as share-ware (FRACTINT is a good example), or as a commercial venture. Some artists may not wish to allow others to use their software at all, on the basis of protecting the 'originality' of their work. We also see examples where the artist adds an interactive front-end to programmed graphics, or just some interaction for specifying parameters that would be tedious to type in as numbers, for example co-ordinates. On older systems you had to write all the code for any interaction yourself, and each software developer did this in a different way, resulting in a wide range of user interface styles on the same machine. A large part of the development effort for a software package would go into the user interface, with the minor advantage however that you could build the interface to suit your perceptions of its design.

Windows 3, the various Macintosh systems, and WIMPS (WIMPS is an acronym for Windows, Icons, Mouse, and Pull-down menus) user interfaces on UNIX platforms such as X-Windows and Motif provide the developer with a standard way to build interfaces, and with libraries of code to incorporate into the software. The disadvantages? Firstly you have to use the interface routines provided, and secondly there is a steep learning curve: Microsoft Windows and the Mac are reckoned each to have about 1000 library functions, and these take time to sort through and learn. With System 7 on the Mac and with Microsoft Windows there is also a whole new philosophy of programming to learn, summed up by 'don't call us, we'll call you'. This means that you have to take into account that the user may wish to use any other part of the system, not just your program, so your program has to be ready to accept messages from the mouse and keyboard that belong to other programs and pass them on. In practice this means that you program has to wait for messages that do concern it, and allow the processor to do other things if the messages don't.

In my own work as a computer artist and educator, I have written about 100,000 lines of code, most of which lies within packages: PolyPaint (an 8-bit broadcast paint system) ICAS, the Integrated Computer Art System, Smart 3D, a simple 3D modeller, and Sculptor, a 3D system based on spheres as the sole primitive. PolyPaint is a system that has been in daily use by myself and my students at London Guildhall University for over five years, and was able to take advantage of locally-available user feedback. I have used one-off pieces of code for special purposes, for example in the backgrounds of some ray-traced Sculptor pieces – see fig. 6 (colour plate) for an example. How though, does the range of affordable new software affect the rational for building software packages in my case? Admittedly, one of the motivations was for research into interface design, but in fact the packages, written from a personal perspective do things that available packages still don't, for example there is little around that makes pattern-making as important part of a package as ICAS did.

In order to collect a range of small maths-based programs together, and also to learn the programming of Windows, I have built a Windows framework for these routines, called 'Mathspic'. The advantage of doing this, apart from being able to collect the maths-based routines together that I like working with as an artist, was to take advantage of the interfacing routines provided by Windows (and also for potentially re-writing the older packages for this environment). Many maths-based programs will take a number of parameters to control, and it is much easier to explore a piece of maths if one changes these interactively, rather than edit one's code, re-compile, and run the program again (consider for example Latham's interactive front-end to Mutator). I have used the Borland Resource Workshop to create dialogue boxes for custom control of my routines (see fig. 7). Fig. 8 shows a custom dialogue box in use to enter parameters for the generation of an image, along with a text panel in a background window with C code showing the calling routine for the dialogue box.

Fig. 9 shows a work entitled 'The Car Crash of G. I. Gurdjieff' which was generated mainly using the C-code routines in Mathspic. The curved patterns are a visualisation of an inverse-square law attraction between gravitational bodies or electrostatic charges, and have been used to show the force-fields between a central body (symbolising Gurdjieff, or any charismatic figure) and those either drawn towards or repelled from him. The six frames are symbolic of the chaos a Master may bring to a community, and also of a final harmony. Other programmed routines allowed the graduated ’embossing’ of the image to the top of the frame, and the selected darkening to the left, neither of which processes would have been easily carried out with PhotoShop (for example), though this package was used in the composition of the program-generated imagery and the scanned images of Gurdjieff and the car. The road was generated in 3D Studio.

In eight years of teaching programming to artists and designers only a small proportion of students have pursued rather than using software packages. Those who have seemed to have had some innate aptitude and sympathy for the approach, but interestingly it seems to have been impossible to predict from interviewing potential students. Perhaps a clearer picture of the visual outcomes could help students decide whether they wanted to take it up or not. I used to consider as a rule of thumb that it takes about 3 years to become a competent programmer, and this is a long period compared to the time it takes to become competent with, for example, the use of PhotoShop as an electronic collage system. One might think that with the advent of plug-in programming things would be easier because others had written the bulk of the code, but in practice the level of competence required to interface with these programs is pretty high. There are also other overheads: in the case of AutoDesk (for 3D Studio and Ani Pro) and Adobe (for PhotoShop plug-ins) one is required to register with the developer’s association for a fee, and to buy specialised compilers. The writing of code for Windows is not for the faint-hearted either; as mentioned before one has to wade through a library of about 1000 routines for the ones that one wants and learn a new programming paradigm. The attractions of Windows, apart from the interfacing libraries, lies with the ability to be independent of graphics hardware (any Windows-compatible graphics card will run any Windows program), cheap platforms, and with the huge potential market for developing a package with commercial application.

Conclusions

The sophistication and complexity of modern computer graphics software means that the need for artists and designers and animators to programme is significantly reduced. However where a set of visual outcomes related to algorithms are required, the writing of programs may be necessary, either in the form of stand-alone code, or as a plug-in to another package. These pieces of code require an understanding of maths and geometry, and considerable programming skill, though the artist may be able to commission the code from a programmer. The idea however that artists universally approach programming through a mathematical or geometrical basis as Franke’s comment (quoted earlier) might suggest, is not born out by the examples looked at in this paper, with perhaps Cohen’s work the furthest removed from this approach. Cohen’s work also challenges the concepts of ‘originality’ in art, a challenge that could not have been made in the same way without the use of programming. Conversely the use of programming can enhance the computer artist’s claim to originality, as in the case of Latham, where the software is not intended for distribution. Latham and Sims have used programming to simulate an evolutionary approach to image-generation, Cohen has used it for an AI approach, while Harwood and Wright, probably representing a wider range of artist/

programmers, have used programming for effects not available through existing packages. In my own work, I have tended to build packages for my own use and others, with occasional writing of one-off software. As a group, the artists discussed in this paper have therefore a wide range of motivations for using programming, which have not been greatly affected by the availability of new packages. However, the group does, in all probability, only represent a small number of artists using computers, most of whom will be well-served by commercial software. We will have to leave history to judge as to which type of computer artist makes the greater contribution to new artforms in this and the next century.

References

[1], [1a], [1b] M. R. King, “Development of an Integrated Computer Art System”, in N. Magnenat-Thalmann and D. Magnenat-Thalmann, eds., New Trends in Computer Graphics, Proceeding of the CG International 1988 (Berlin: Springer-Verlag, 1988) pp. 643–652.
[2], [2a], [2b] M. R. King, “Towards an Integrated Computer Art System”, in R. J. Lansdown and R. A. Earnshaw, eds., Computer in Art, Design and Animation, Proceedings of the 1986 conference at the Royal College of Art (London: Springer-Verlag, 1989), pp 41– 55
[3] M. R. King, Computer Media in the Visual Arts and their User Interfaces, unpublished doctoral thesis, Royal College of Art, London, 1986.
[4] H. W. Franke and H. S. Helbig, “Generative Mathematics: Mathematically Described and Calculated Visual Art”, in Leonardo, 25, Nos. 3/4 (291–294) 1992.
[5] C. Lindley, Image Processing in C
[6], [6a] G. Harwood, IF Comix Mental, Working Press, London 1989.
[7], [7a] H. W. Franke, Computer Graphics – Computer Art, Phaidon, 1971.
[8] J. Whitney Snr., Digital Harmony Peterborough, NY McGraw-Hill 1980.
[9] D. Em, The Art of David Em, New York: Abrams 1988.
[10], [10a] S. Todd and W. Latham, Evolutionary Art and Computers, Academic Press, 1992
[11] J. Pearson, “The Computer: Liberator or Jailor of the Creative Spirit”, in Electronic Art, supplement issue of Leonardo, 1988, pp. 73–80.
[12], [12a] P. McCorduck, Aarons Code – Meta-Art, Artificial Intelligence, and the work of Harold Cohen, Freeman, New York 1991.
[13] Mark Wilson, entry in Electronic Print, an International Exhibition of Computer Art, Arnolfini Gallery Catalogue, Editor Martin Reiser, Bristol 1989. See also the inside cover of Digital Visions, reference [15] below.
[14] E. Zajek, “Orphics: Computer Graphics and the Shaping of Time with Color”, in Electronic Art, supplement issue of Leonardo, 1988, pp. 111-116.
[15] C. Goodman, Digital Visions, Abrams, New York, 1988.
[16] K. Sims, “Panspermia”, in Eds., C. Schopf and M. Knipp, Der Prix Ars Electronica, International Compendium of the Computer Arts, Veritas Verlag, Linz, 1991, pp. 76-83.
[17] Harold Cohen, Tate Gallery catalogue, Tate Gallery Publications, London, 1983.
[18] John A. Walker, Glossary of Art, Architecture and Design, 3rd Edition, London Library Association, 1992, London.
[19] R. Q. Wright, “Superanimism: the practice of formalised imagery” in Computers in Art

and Design, Editor Isaac Victor Kerlow, Association for Computing Machinery, New York 1991, pp 85-88
[20] R. Q. Wright, “Some Issues in the Development of Computer Art as a Mathematical Art Form”, in Electronic Art, supplement issue of Leonardo, 1988, pp. 103-110.
[21] M. R. King, “Sculptor: A Three-Dimensional Computer Sculpting System”, in Leonardo, 24, No. 4 (383-387) 1991.
[22] Cass and Miller ‘Rapid Stable Fluid Dynamics’ in SIGGRAPH conference proceedings Vol 24, No. 4, August 1990, pp. 49-57.

This paper is published in Leonardo, Vol 28, No.2, pp. 113-121, 1995. Permission to reproduce this paper is kindly granted by the author and Leonardo.

Dr Mike King is Reader in Computer Art and Animation at London Guildhall University.