Programming Literacy for Artists

McCANN, Karen
City University of Hong Kong
smash@cityu.edu.hk

Abstract

If you, the artist, speak computer, you can control your data. That's all that computer languages do – manipulate data: words, pictures, sounds and other programs. Many artists can see the advantages of computer worlds with complex emerging behavior but can't find the way in. The way in is with high-level languages used with an immediate relevance to the realization of an artistic idea. Literacy for electronic artists is fluency in programming languages and their digital interconnections. This paper proposes a methodology for the electronic artist wishing to come up to speed.

1. Programming Interactivity in Art

"With the advent of new computer technologies new dimensions were introduced to the artistic creation process: virtual space and real-time interaction." [1]

New media art is interactive – that's what makes it new. Interactivity is not new in itself. People see a different movie even when we sit side-by-side, different images in our heads when we read the same book, different strategies when we play a game. This kind of interactivity is the imagination space the viewer/reader/player constructs within the story, but this is a limited kind of interactivity. Deeper interactivity in basic human actions like drawing, conversing, playing music, calculating, stems from the human need for self-expression.[2]

Enter the digital age, where the computer can input all sorts of data from sensors, it can output all sorts of media, it and it can process the inter-relationship between them in fast, clever ways – as clever as we can make them. But are we clever with new media art? Not as clever as we could be. The majority of interactive installation art today is dominated by first-order interconnections between input and output. It most often consists of sensors triggering pre-captured media according to the artists' plan - *reactive* art.[2] The intent, meaning or comment of the artwork can still be strong in these 'first order' artworks, however, they don't harness the capabilities of the computer – the ability to engage in real-time processing.

The most interesting commercial software for digital media incorporates built-in scripting languages, which extend the possibilities inherent in the software to real-time interaction with input devices. Scripting languages are a good place for artists to start extending their skill sets. Artists learn the power of variables, consider the control structures ruling the computer, cope with the exacting nature of writing in code and appreciate the processing power of the machine in front of them to produce 'reactive' media artworks.

Scripting is a good place to start – but it is not real programming. When the artist finds something that the software just can't do – then they are ready to make the leap into code.

2. Making the Leap

The easiest way for artists to use programming is to make friends with a programmer. Artworks that push the boundaries typically come from collaborations with computer scientists and these works are often closely linked to research labs using complex proprietary software. It is a rare and wonderful breed of computer scientist who embraces art. However, making the leap from artist to programmer is perhaps even harder, due to complex personal, societal, perhaps even neurological issues.

So is there is a magic line between those who can program and those who cannot? Programming is not a natural-born skill. Learning a computer language is similar to learning a foreign language – you can speak just enough to get around, or you can attain the level of native speaker. The point is that even enough language to 'get by' is invaluable from the perspective of wider communication. The more you understand how the computer thinks by trying to talk to it (through code), the more you can understand its capabilities and its deficiencies. Fluency is, of course, entirely separate from having something worth saying.

3. Working Against The Fear Factor

"The kind of mathematics *foisted on children in schools is not meaningful, fun or even useful."* [3]

Societies, over the past century have arbitrary divided art and science. The artistic personality is sensitive, given to abstraction and good at visualizing ideas. The mathematical/scientific mind (in contrast) is logical, particular and adept at numerical calculation. Teachers are 'trained' to recognize these 'types' by secondary or even primary education level. Universities, carrying on this tradition, are divided into Humanities and Sciences.

Art has always borrowed from science, whether it is through the painter's technique of perspective, or more modern digital borrowing. In the last decade there has been an about-face by many in the artistic community. As late as 1995 there were artists proclaiming proudly that they didn't have an e-mail address or own a computer. These voices have all but vanished.

Although more people are familiar with software, few people really understand what is going on beneath the surface. People often don't realize that the GUI is an interface, 'windows' are an invention and 'cutting and pasting' is a learned technique. A generation has been born who never knew life before the Internet.

4. A Teaching/Learning Methodology

The standard approaches to learning computer languages may have to be different for the artist. Most books selling the premise of learning a language in 20 minutes are either so basic that they never get to the good stuff, or so jargon-filled that they are inaccessible to the non-technical mind. But which language to choose? The short answer is any one will do, as they are all essentially the same. The longer answer is to choose a language

that is suitable for your goals. If you are a web developer it makes sense to choose Java, if you want to extend your computer animation skills you could choose Perl or even C. Most commercial software is written in C or C++, but that doesn't mean that these languages are appropriate for art. I teach the object-oriented language Python because it's cross-platform, high-level, the available documentation is fun and it's free.

4.1 Low Level Languages

There is a hierarchy of programming languages when you speak to a computer. At the very lowest level, the bits and bytes, transistor level, the machine only understands bursts of electricity. Built above this is a layer of 'machine language' that consists of a long stream of simple instructions like start and stop and jump and includes pointers to addresses in the memory. It is long and boring to read. At the middle level are the more complicated languages such as BASIC or C. These languages have libraries of common functions that do things like read or write files and have to be translated down into machine language for the computer. The code is easier to read than machine language, but they're harder to learn than the latest high-level languages, because the higher you go up the hierarchy the closer the commands are to natural English language.

4.2 High Level Languages

The most popular of the higher-level languages are object-oriented languages like C++, Perl, Java and Python. Lower-level languages like C have to be completely compiled into a machine-level translation before they are run. These languages are typically translated on the fly. This means that you can interact with just little bits of code to test your ideas. For example, you can say x is 3 and y is 5 and then ask what x + y equals. It will give you the answer back immediately – which is very good for quickly debugging code.

The 'object-oriented' name refers to a control-structure idea. Object-oriented programs are written as a set of inter-related mini-programs, a modularity where you can use the same program in different ways through their interconnection. Object-oriented programs allow interesting behavior to arise because the modularity of its construction allows you to think on a higher level without having to worry about the serial order of commands. It is not necessary to learn this level of the language when you start out, but their more advanced structure allows you to reuse and rearrange pieces of your code in completely new ways later.

5. Starting Out

"*… an algorithm is a fail-safe procedure, guaranteed to achieve a specific goal.*" [4]

It is essential to have a creative goal in mind when you start to program. One important point is to have something fun to work on – otherwise it be a chore to learn. Don't be too ambitious and don't give up when it doesn't work the first time, or the second, or the hundredth time.

The process of debugging your programs is the most beneficial of the programming experience. Computer problem-solving skills can often be adapted to obstacles outside the programming environment. However, writing a program is not like writing an essay. It is usually not possible to stay up all night and have something, anything, resembling a finished work. The way to attack a programming problem is to break it down into pieces, simplifying it, being systematic, and approaching it from different angles.[5] This means starting and stopping frequently, taking breaks and walking away to come back refreshed. This is another practice that can be beneficial outside the programming environment.

6. Down to Specifics

Variables. Control Structures. Data Structures. Algorithms. Understand and use these four elements and you are a programmer. These four areas of programming will be met immediately and simultaneously when you start programming – division between them is artificial.

6.1 Variables

Variables are the place to start and they are very powerful ideas. If you can remember back to high school algebra and x+y=3, you have already used variables. However, they are used in a different way with computers – as placeholders. They can hold numbers, series of numbers, commands, words – even other variables. The important thing to remember is that they have a function.

Variables should be named according to their function, things like "counter" or "table" or "keepRemainder".[6] When I first started to program I called my variables things like "John" and "Sarah". But guard against using any of the special words the language reserves for commands. Try to keep the names short and if they recur frequently use just a letter like "i" for "counter".

Counters are particularly useful ways to use variables. Consider the common line of code $i = i + 1$. To an artistic type this line is incomprehensible. How can i equal anything other than i? We can understand $i = 3$, but $i = i + 1$? This not designed to confuse, as it is not an equality statement, but a right-to-left assignment. It is often the last line inside a loop opening up possibilities for recursion, leading to cool things like fractals.

6.2 Control Structures

Loops are one of a number of control structures. Historically (and controversially) attributed to Charles Babbage's colleague Ada Lovelace, the loop is a fantastic idea for using the remarkable processing power of the computer. Computers are very good at repetitive procedures and loops are an essential technique. It is a very simple one and would count as the first of many control structures to learn. Control structures range from low-level loops to high-level object-oriented coordination.

Take a look at your creative goal and use a flow chart to plot how the data will be manipulated. Break it down into pseudo-code so that each section is described in English, but in the way a computer would think about the data flow. Finally break it down into real code, testing each little bit as you go.

6.3 Data Structures

Data can be in variables, or it can be in a list, or a dictionary where one value equals another, or in an array or matrix where data can be accessed in a method like Cartesian coordinates. Data can also be written out to a file. All of these methods of storage and retrieval have to be practiced in order to achieve proficiency with programming.

6.4 Algorithms

Algorithms are the business end of programming. Algorithms are mathematical methods for sorting, searching, processing and filtering – essentially they are proven techniques for working out a problem. An example would be an algorithm for a pseudo-random number generator, a very handy program. Algorithms for AI and neural networks are at the cutting edge of technology and they are often beautiful and inspiring in their simplicity.

7. Getting there from here.

Whichever programming language you choose, you may struggle initially with the syntax as you try to bend your idea into 'computer-speak'. The syntax is unforgiving, which makes it both harder and easier to learn than natural languages – harder in that if you misspell even one word your program will not work; easier in that there is no grey area with computer languages, the computer will only do what you want it to do.

There are lots of advantages to learning programming. It helps you even in everyday speech say exactly what you mean – and in the right order (as in giving directions). If you work with programmers in the future (and good friends they are), you can understand where they are coming from. The biggest advantage of learning programming is that it offers new opportunities for making art. It broadens problem-solving skills and opens your eyes to the real capabilities of the ubiquitous imagination machine.

Reference

[1] Sommerer, Crista and Mignonneau, Laurent "Art as a living System", Art @ Science, Sommerer, Crista and Mignonneau, Laurent (eds.), Springer-Verlag/Wien, Austria, 1998.

[2] Stern, Andrew "Deeper Conversations With Interactive Art, or Why Artists Must Program" in Convergence: The Journal of Research into New Media Technologies, Spring Vol. 7 No. 1, 2001.
http://home.netcom.com/~apstern/interactivestory.net/papers/deeperconversations.html

[3] Papert, Seymour, Mindstorms (2nd Ed.), Basic Books, New York, 1993.

[4] Hillis, W. Daniel, The Pattern on the Stone: Simple Ideas That Make Computers Work, Weidenfeld and Nicholson, London, 1998.

[6] Polya, George, How to Solve It, Penguin, London, 1990.

[5] Kernighan Brian W. and Plauger P.J., The Elements of Programming Style (2nd Ed), McGraw-Hill, New York, 1978.