# Expression and Time: the question of strata and time management in creative practices using technology

Thor Magnusson

Let us rewind to the late 1990s. In the West the internet had become ubiquitous and artists were drawn to the new potentials of artistic expression through the medium of the computer, allowing for interactivity, collaboration and connectivity across time and space. This resulted in the emergence of various fields such as net art (where the HTML code is used as artistic material), software art (where software is re-interpreted or commented on from cultural and political standpoints) and robot art (where computers are used to control hardware bodies, often through telepresence). It also injected new life into fields such as interactive installations, video art and sound art. The problem here was that of tools. There were not many tools that allowed for sketching or fast generation of ideas and producing complex works that both look good and work well. The artist could either learn languages like C/C++ or Java and try to code their own engines for artistic creation or decide to use higher level environments such as Macromedia Director that provides a graphical working environment for multimedia production. Director contains a scripting language (Lingo) for more complex interaction or networked data communication. The limitations of Lingo defined the aesthetics and the nature of the work that could be produced.

Fast forward to the middle of next decade. We are now faced with a panoply of free open source tools and programming environments that have been created by artists-technologists for other artists-technologists to use. The explicit aims of these environments are to function as open toolboxes that do not limit or direct the artist regarding form or content (Puckette, 2002; McCartney, 2002). From the level of analogue input into the machines (Arduino, Wiring) to operating systems (pure:dyne, Planet CCRMA), to programming languages and frameworks (SuperCollider, Pure Data (PD), ChucK, Processing, OpenFrameworks, etc.) to distribution media formats (flac, ogg), we are seeing proliferation of environments made by artist-technologists to create and distribute their work. The aim here is not only to create free and open source (thus collaborative and

sharing in nature) production tools, but more importantly to allow for tools that do not necessarily limit the artist's expression in rigid ways. For example, when a user stumbles into an expressive limitation of Director, 3D Studio Max or Cubase, there is no way of getting around that limitation. Even when a bug is found, the user might have to wait two to three years before it is fixed, although updates to the software have happened in the meantime. In the situation of free open source software, the user can fix the bugs and extend its scope with very little effort. Often a mail to the mailing list describing the problem situation will have it solved within few days. Most importantly, open source tools give the user the feeling of really owning their creation, as they are able to understand and relate to the tool they are using.

This is a true paradigm shift and a very positive one for people using the computer for creative work, whether it is music, graphics, video, multimedia, installations or robotics. However, there are a few drawbacks, typically those related to the effort required to learn the environment. Working in open source programming languages can be time consuming and lead to diversions from the creative process where the production of the artwork sometimes becomes tangled up in working on the environment in which the artwork is made. The relationship between work and its foundations blur. This is not necessarily a negative. The problem is that of time-management and how artists manage to situate themselves at a level of code that benefits their work. In this paper we will look at the reasons why working with open source programming environments for artistic production is a meaningful and rewarding activity, despite its dangers regarding the expenditure of time.

## Why Open Source Programming Environments

Above I described the situation in the mid 1990s, where there were few alternatives for art practitioners to create their works unless working in quite time consuming and unfriendly programming languages like C/C++ or Java. The computer had become fast and cheap enough for use in interactive installations, algorithmic composition, real-time synthesis of audio and video, and powerful digital signal processing for manipulation of sound and graphics. As artists are normally inquiring people who want to know why someone designed the tool to perform certain ideas and not others, there was a need for environments where artists could explore the cogs and wheels of programming as artistic material which cannot be done if the source is hidden.

This need is expressed by Jaromil the creator of dyne:bolic: a Linux distribution that includes all the main tools a media artist might need:

> ***dyne:bolic*** *is shaped on the* needs *of media activists, artists and creatives as a practical tool for* ***multimedia production****: you can manipulate and broadcast both* ***sound and video*** *with tools to* ***record, edit, encode*** *and* ***stream****, having automatically recognized most device and peripherals: audio, video, TV, network cards, firewire, usb and more; all using only free software! [my italics] (Jaromil, 2007)*

We read from the creators of pure:dyne, also a Linux distribution with a specific focus:

> *The development of pure:dyne can be traced back to the inclusion of Pure Data in the dyne:bolic liveCD distribution. As this addition became increasingly popular, there was suddenly a* demand *to increase its support for Pure Data in a more serious production context... Today pure:dyne gathers a growing user community and has been used in numerous workshops and performances. [my italics] (Mansoux, Galanopoulos & Lee, 2007)*

And the maintainer of Planet CCRMA (a Linux distribution including all the main tools for sound and video production) describes how users outside the Stanford University CCRMA centre began to use the Planet CCRMA as the link to the system circulated and search engines pointed people to its existence:

> *This changed the nature of the project. As more people outside of CCRMA started using the packages I started to get* requests *for packaging music software that I would not have thought of installing at CCRMA. The number of packages started to grow and this growth benefited both CCRMAlites and external Planet CCRMA users alike. [my italics] (Lopez-Lezcano, 2005)*

All three operating systems – dyne:bolic, pure:dyne and Planet CCRMA – respond to certain needs and demands in the general culture. All three have become immensely popular and are used all over the world. The situation has changed dramatically: whereas a decade ago an artist working with computers would probably work on a proprietary operating system and with expensive software – for example, Director, where a three year old version of the software sets you back by over £1000 – users now have incredibly cheap hardware, free operating systems and many of the best creative environments are open source and free. The impact is huge and more so for cultures whose currency exchange rate is not favourable to the Western valuta, rendering the chance to pay legally for software all but impossible.

Many of the people working in the field today started using computers when, in the middle of the 1990s, they became available in an affordable price range and interesting enough for creative purposes.[1] They either had to learn the hard way with low level languages that required too much time operating on a level that was more computer science than art. Or they would be stuck in the constraints of high level packages that limited their creativity. They began to collaborate on projects, distribute code and very often this resulted in environments that allowed others to jump the step of computer science and get involved creatively straight away. Here is how the creators of the Java-based programming language Processing construe the *raison d'etre* of the software:

> *Processing relates software concepts to principles of visual form, motion, and interaction. It integrates a programming language, development environment, and teaching methodology into a unified system. Processing was created to teach fundamentals of computer programming within a visual context, to serve as a software sketchbook, and to be used as a production tool. Students, artists, design professionals, and researchers use it for learning, prototyping, and production. (Casey & Fry, 2007)*

As this book attests, the FLOSS (Free Libre Open Source Software) movement in the arts has become strong and functional. There now exists free and open source software for almost everything one would like to do and often it is impossible to find the equivalent tools developed in the commercial sector. Tools like SuperCollider, PD, CSound, Ardour and Audacity for music; Processing, openFrameworks, Fluxus and Mirra for graphics; Wiring, Arduino and MUIO for hardware interfacing and countless open game engines for 3D gaming, film-making or interactive narratives are all providing the artist with highly professional, expressive and focussed tools for the task.

But things are not so simple. If we look at the fields of music or the fine arts (at least up until modernism) we see that much art involved the development of fine motor skills, a craftsmanship where the trained body would produce works of virtuosity. Today, when working with automated and intelligent machines, we do not train our bodies any longer. The training consists in the comprehension of the underlying systems of the media we are working with and their modes of production, distribution and consumption. It becomes a question of understanding the field of software and their multiple cultures, of choosing an environment at the right level in which to work. Yes, the media arts are quite cerebral by

[1] The precise timing here is obviously debatable: some might say with the Atari computer and the invention of MIDI but others would argue that it was when one could perform real-time manipulation of audio and video in high resolution.

nature, but much of modern art since Marcel Duchamp is. Virtuosity has taken on another meaning: it now involves a deep understanding of computer science and the science of digital signal processing and mathematics with regards to sound, visuals or 3D spaces.

Learning a programming language is not an easy task. One can become conversational in a month or two, but fluency of expression is something that might take over two or 3 years, with many hours put into the practice every day. Further, the logic of programming is not the only thing that has to be mastered: working at this level means that the artist has to understand the nature of sound and visuals from both physical and psychological perspectives. (In the case of music one would learn the physics of sound, digital signal processing and psychoacoustics). This takes roughly as much time as mastering an acoustic instrument. The problems here are often related to the power of the computer as a medium. The computer and the programming paradigms that exist for it are strong conceptual models that prime the mind of the student into particular ways of thinking, perhaps altering the cognitive style of the student. The question becomes: what is the nature of the tools that we are currently using and how well do they reflect the demands of artistic practice? Does the artist-technologist create the tool or does the tool create the artist-technologist? To what degree will the artist develop a different cognitive style through studying computer science?

**Time and Expressivity: how/why does this equation work in practice?**

Developing an operating system, programming language or a sketching environment for creative work is time consuming. For one creative person this can be too much sacrifice for what the initial idea involved. For another, programming is a mode of thinking and as art is essentially about thinking it becomes the tool *par excellence* for artistic creativity.

Above, we talked about the strata of programming abstractions and how each artist chooses his/her tool for expression with regards to what he/she wants to do and how. For the sake of pseudo-science and from an urge to have an equation in this chapter, an illustration of this problem might look like this:

$$\text{Creative product} = \frac{\text{time}}{\text{expressivity of environment}}$$

That is, the more expressivity we have through the capabilities of the environment, the more time it takes to make the creative work. This has nothing to do with the aesthetic quality of the works, as that cannot be defined by the technology used to produce them.

In order to understand the question of time management and expressivity, i.e. the question of making the tool itself versus making something using the tool, I decided to go beyond my own personal experience of these things and ask a few colleagues who are already practitioners – developers, programmers and artists all at once – about their work patterns and attitudes towards time and expressivity. How do creative people justify their spending time on building tools and not the artwork itself? Why the fascination with code? How can this technology represent our thinking? From the replies to the survey, we can detect a few threads originating in the idea of code as creative material.

**Coding as Self-Understanding**

*What I cannot create I do not understand – Richard Feynman*

One of the participants in the survey quoted Feynman as above. This is an illustrative idea that most of the participants seemed to subscribe to. The desire to learn about one's music or visual art through building the tools is a common characteristic of all the practitioners. A crucial distinction is whether one is building the tool for a personal or public use. The latter seems to be time-consuming on an exponential scale.

> *Tools have such a strong impact on the artistic process that I like to know mine well and there is no better way to know a tool than to build your own. Paul Lansky once said that he didn't really distinguish between building the tools and making the artwork. I have a lot of sympathy for this idea. [AS]*

> *Where this question becomes tricky for me is in relation to developing tools for other users. On the surface this really is a black hole. You end up spending a lot of time supporting other users needs and this generally has little to do with your own artistic practice. However, end users are really good at forcing you to do the bug fixing, stability and performance improvements that you would probably not do on your own, but which you are really glad you did! [AS]*

**No Distinction of Building a Tool and a Artwork**

Most actions require creativity unless one has incorporated some skills for rote activities. A recent survey (Magnusson & Hurtado Mendieta, 2007) showed that many musicians actually thought playing an acoustic instrument was a cliché-prone activity as it was so practised and so

inscribed in a cultural tradition. Building one's own tool is a highly creative activity that forces a self-awareness in the artist and which often cannot be separated from the process of making the art.

> *In software art... It is impossible to dissociate the creation from the process, to the point where using the words 'tool' or 'environment' becomes completely irrelevant. [AM]*

> *Programming for music is a part of music-making, just as ruling bar-lines on manuscript paper can also be a modest part of music-making, all distinctions less important to my mind than whether or not one is concentrated when working. [TH]*

> *I rarely use a piece of software I've written for more than one thing, so use and development are the same in most cases. I'm not really even that bothered about the 'music' as I don't really see it as the primary goal or validating factor. [TB]*

### The Artwork as a Process

Software art fits well with the modernist tradition of seeing art as a *process* rather than a finished piece as software is not written in stone. Software is never finalised. Only the material constraints of older art forms reinforce our habit to see art objects as completed works of art. These are constraints that were introduced mainly with the Gutenberg press and the phonograph. Before these media technologies, texts and music would normally change every time someone wrote another copy of the text or played the song. Code as immaterial, abundant, copy-able, omnifunctional and executable text is never dead. Instead of concluded pieces of art 'frozen' in time, we have versions.

> *If the art is truly living in the process, then developing it will not only change the guts but also influence the output as they are the same and sole object. [AM]*

### Coding as a Conceptual Practice

Art can be highly formal, mathematical or scientific. Music is a good example of an art form where practitioners have investigated formal and mathematical relationships in harmony, melody, timbre and meter. Programming as a means of formalising one's thought, externalising them and testing them in performance using a logical machine is therefore a tempting method of working for many artists.

> *Programming problems can be an interesting thing to pursue while musical problems (as defined by Schoenberg and others) are being turned over in one's mind. Especially so as programming problems thus encountered are usually simpler than the accompanying musical problems. [TH]*

> *For me art is not so much about expression, but more about reasoning, so the process is maybe a different one. The environment is not separate from the work, neither as a work of art nor as a work of theoretical research. [JR]*

### Creating the Tool for Originality

The modernist demand of originality in our art is still with us and many people consider the highest form of creativity to be artistic creativity where the artist transforms the cultural space in which he or she works in (Boden, 1990). For many, it is by necessity that they build their own tools, as using other people's tools might lead to less thought and original design solutions to both aesthetic, formal and technological problems.

> *Inevitably tech issues intrude. But I guess you want to come up against difficulties; else you're not pushing the boundaries? [NC]*

> *Knowing that I'm using an obscure programming language that very few are using makes me excited, as the language allows me to think and compose in certain innovative ways. [AN]*

### Coding as an Artistic Practice

Many of the participants in the survey see coding as an artistic practice in itself. In fact some of them see it as a performative act and 'live-code' in front of audience with their screens projected on the wall (Toplap, 2007; Nilson, 2007). Coding is here seen as a way of externalising thoughts, in a manner similar to sketching by drawing or model building. When programming is seen as a performative action, a choreography of thought, it ceases to be a means to an end and becomes an end in itself.

> *I see programming as a part of my creative practice, just like playing the piano or studying books on composition. As such I try to be as fluent at it as I can afford to be so that my creative ideas can be realised as efficiently as possible. This is not at all to say that programming is transparent to the creative process – like any tool it has an impact on the work. [AB]*

## Coding as Craft: An Inscribed Skill

Any search engine will show countless books and articles on the topic of coding as craft; a skill that has to be mastered through time with lots of practice. The programmer learns to think in the language that he or she works in and formulate the problems in the terms of that language, often conceptualising the world through the means of programmatic paradigms. But coding is not only a craft: it is a performance as well, as the case of live-coding exemplifies. Live-coding requires that the practitioners are good at their trade, which involves fast thinking, fast typing, practised algorithms and good knowledge of the material (programming environment) they are working with. (Nilson, 2007; Sorensen & Brown, 2007).

> *I find my expressiveness is limited by the 'gestures' my environment can support. I prefer to be able to express things in my library very tersely, so I see working and thinking about changes to the performance environment as relevant part of my time spent. Much as an instrumental performer must spend time learning new techniques, the live-coder must take time to extend their library of musical gestures. [GC]*

> *A virtuoso computer musician needs to be a decent programmer... [AB]*

## More Fun Building the Tool Than the Work

The concept of art is famously narrow and defined by cultural practices. Most of the participants of the survey were not concerned with 'art' as an isolated cultural phenomenon and saw creativity as a ubiquitous human behaviour. For some there was not only the absence of distinction between the tool and the work, but building the tool itself was more important… and fun:

> *Building sequencers rather than sequences seemed to be more fun. [IZ]*

> *For me, the software itself is what I'm focusing on and what I like to present as my work. I sometimes perform with it and have released some recordings of the tool in practice but those almost have the status of a 'demo'. I find that some people don't understand this attitude, but that's not really my problem. [AN]*

However, some wanted to draw a distinction between making the tool and the work that could be made with the tool. In fact this seemed to be related to the level of coding in which the artist is working. For example in

PD one could write an external (an object encapsulating more complex and lower level code) or in SuperCollider one would write a Unit Generator. This is a stratum down from patching in PD or coding in sc-lang (the native programming language of SuperCollider) as it involves more debugging and, most annoyingly, having to compile the code before putting it to use.

*I think it is what it is. If you develop a UGen, you develop a UGen; you shouldn't kid yourself that you're making music, though you might be making an essential technological component. But things blur again, for example, in testing the UGen, which is suddenly so very exciting, musically... [AN]*

**The Danger of Spending Time on the System not the Art**

Most participants acknowledged in some way the danger of forgetting oneself in building the ultimate creative system and never have time to use it for creative work.

> *Creating your own ultimate dream system to make art is a dangerous game as most of the time the focus is attracted by the building of the environment itself, developing it, updating it, documenting it and never ending adding features to it. It can become so important that the art produced with it becomes a minor manifestation or demonstration of the system. [AM]*

> *Sometimes I rehearse every week, but this is perhaps for 3 hours, vs. 30 hours working on the environment. [RB]*

This distinction however is rather superficial as there is as much creative thought put into the design of an environment as the design of a musical piece for example. People have started seeing the building of tools as an artistic endeavour and Linus Torvalds getting first prize in Ars Electronica 2003 for the kernel of his Linux operating system is good proof of this.

> *It is not easy to give an artistic statement about an operating system, because while an operating system can be a work of art (I certainly feel that there is an artistic component to programming), it's not in itself very artful... In more 'artistic' terms, you might consider the operating system to be the collection of pigments and colours used to create a painting: they are not the painting itself, but they are obviously a rather important ingredient – and a lot of the great painters spent a large portion of their time on making the paint, often by hand, in order to get their paintings to look just right. (Torvalds, 2003)*

## Coding Takes Time

[2]
http://www.signwave.co.uk

Most of the participants seemed to have a stoic attitude towards the time they put into their work. They see this as the nature of art itself; in order to make good art, you have to work a lot whether that is through reading, preparing materials, learning, exercising or talking to people. Programming and learning the skill is no different than learning how to play an acoustic instrument or train the hand in freehand drawing.

> *Well I am still using up the mileage to be gained by convincing yourself that all the time you spend screwing around with open source software trying to get it to work makes you learn things you wouldn't have learnt otherwise. [MYK]*

> *When preparing for a piece, there is always a trade off between what is possible within the environment, what do I need to add, and what am I able to implement (and debug!) before showtime. [MB]*

> *In a way, the 24h to the day limit seems like an inconvenient constraint; but the again it is good preparation for the limited lifetime constraint we all face at some point, and trying to spend that time well. [ADC]*

## Programming as Meta-Art

Creating something that creates something else is on the meta-dimension. Many software artists, such as Adrian Ward, see themselves as creating meta-art, i.e. art that is used by artists to create more art.[2] It varies how the programmers see their role, from being participant in the creation of the end object (as Ward claims) to rejecting all co-authorship and clearly separating the software as meta-art and the product that it creates when an artist uses it.

> *As I see it, if I create a tool that is used by artists in their work, I have created something that is of value to them as a product. I see this as an object of art and the artists using my tool not only as users but also as aesthetic 'actors' or 'perceptors' for lack of better words. (Basically for what 'listener' is to music, 'viewer' to film, we need a word for the person that engages with software art). [AN]*

### What I Cannot Take Apart, I Don't Own

[3] http://makezine.com/04/ownyourown/

Much commercial music software sets up the whole structure of the music by default and the process of composing is more a question of removing presets than composing from an empty plate. Many people find that what they cannot take apart and understand the bits of, they cannot claim is their own creation. This idea can be found in circles of people working with hardware just as software.[3] But the question is not only that of understanding, but also of archiving as commercial formats are often closed. An artist that produces work in a closed source format or software might not be able to revisit that work in the future.

> *I feel I need to understand the technological foundations of my music, what cogs and wheels I am using to implement my ideas and how they in turn change through using the tool. Therefore open source software is important to me, if not I'd be stuck in a situation where I feel I am a mere consumer of the software and not an active participant. Of course, this has to be taken with a grain of salt, as there is obviously code 'all the way down' and I'm not interested in looking at machine code or processor design. [AN]*

### Oh, Where has the Body Gone?

The problem of embodied skill, the trained musical/artistic body, is something that is ingrained in the question of coding as artistic practice. This feature of the man-machine relationship has been dealt with in recent survey (Magnusson & Hurtado-Mendieta, 2007) but it is something that comes up repeatedly when talking about the digital arts. For live-coders, the haptic interface of the computer (such as mouse and keyboard) might just be an unnecessary interruption between the thought and the implemented algorithm (hopefully to be solved soon), but for other artists, the body is an integral part of the creative process. The pre-linguistic or unconscious mind might be better expressed through bodily movements than as linguistic thoughts after all.

> *Also sometimes I find it quite cumbersome to have to program everything, rather than being able to just play, like I can on a piano or an analogue synthesizer; i.e. instruments that have behaviour in themselves. [MB]*

### The Question of Strata and Location

As seen from the preceding chapter, programming is a craft, an inscribed skill that involves complex relationships of an environment and its logic expressed in a programming language; the programmer and his/her

integration of the environment's logic; the programmer and his/her embodied relationship to the hardware used; and various other threads that connect the finer elements of the human-machine network. The question for the artists is where to locate themselves in the strata of expression through code. Where in the machine building process do they want to work?

Here it can be useful to think of software as strata: as concrete layers or sedimentations of varied abstractions. Assembler is an abstracted machine language; Java compiles into a virtual machine that is machine language; SuperCollider is a language written in C++ that is a level above assembler, etc. These are all strata that function on their independent level, but allow for modifications across the spectrum (or belts of abstraction) if we are working in open source.

> *The strata are extremely mobile. One stratum is always capable of serving as the substratum of another, or of colliding with another, independently of any evolutionary order. Above all, between two strata or between to stratic divisions, there are interstratic phenomena: transcodings and passages between milieus, intermixings. Rhythms pertain to these interstratic movements, which are also acts of stratification. Stratification is like the creation of the world from chaos, a continual, renewed creation. (Deleuze & Guattari, 1987. p. 502.)*

If we look at software as the phenomena that Deleuze and Guattari describe, it becomes fundamentally wrong to sacrifice your freedom of thought and action to a closed source environment, and we will see why in the third section of this chapter.

We have already seen from the threads above, how some people enjoy the building of a tool even more than performing or composing with the tool itself. The focus ranges from working on operating systems, the source of an expressive language (like creating the source code of SuperCollider, PD or Processing) or working with code as expressive means and try to shy away from computer engineering tasks. Each programming language has its own characteristics and for some it might be better to work in a high level language like Lingo or ActionScript where others click with Processing or SuperCollidier. For yet others, thinking in a graphical programming language like PD is easier and more intuitive, and switching to writing objects for it (at a lower level) is only done by necessity. An important question here is that of the economy of time vs. expression. Many people are ready to invest more time in the building of their expressive environments to acquire in return personal, unique and aflexible methods of production.

However, this is not the whole story. Choosing to work with PD rather than Max/MSP or Processing rather than Flash or Director is also a political decision. It relates to the freedom of individual expression; who is in power; relations to the capitalistic market; the question of open formats for future archiving; and a clear awareness of what it means to be a user of software.

## Ideologies in Software Environments

An extensive cultural critique of software as expression is yet to be written although Matthew Fuller (Fuller, 2008) and a few others have done excellent job in trying to create the discipline of software criticism in the way such discipline exists for literature, film and music. The history of Human Computer Interaction (HCI) has gone through various phases and a few of them are relevant here.[4] The initial and naïve way we saw the functionality of software was that of a neutral symbolic parser that would take data input, perform some logical calculations and then output some other data. The basic assumption here is that the mind and its processes are independent from any material manifestation or embodiment. Actions are seen as logical, pre-planned and well definable in the context of the task to be performed. There is a separation of the internal world of the subject from the external world of objects. The agent is always a rational agent that moves in a world of logical laws. All representation of the world or understanding of it can be expressed in logical formulas of which the computer is the machine par excellence. The early Wittgenstein would be a good example of this way of construing the world.

But as the later Wittgenstein discovered, this world-view is a simplistic portrayal of the human situation and forgets about the embodied nature of our thinking and perceiving our environment. The fields of hermeneutics and phenomenology bring us a clearer understanding of what it means to be a human in the world and interpret the signals around us. Nothing is neutral any longer. As designing a programming language or a tool necessarily involves taxonomic design, it becomes an ontological endeavour. This is a question of transformation: how an individual system designer interprets the world and represents it in a cultural computational engine that is necessarily political as well as aesthetic.[5] For an artist, the idea that someone has looked at the world and categorised it into sections that are represented in the functional aspects of a software can be simply uncanny. To what kind of ideology am I subscribing when I use this particular operating system, this software or this interface? These are

[4] Here we should read HCI as literally as possible, i.e. not as a field concerned with how a button should look or function, but as a field that should study the more philosophical questions related to what it means to be a human that deals with a symbolic machine created by the human itself implanted with its creator's symbolic logic

complex questions with no clear-cut black-and-white answers, but they are questions that many people deal with albeit in an indirect manner when they choose their tools.

[5] Amazingly little has been written about the way commercial software companies effectively run a new type of Western imperialism with their way of representing thought and activities through their interaction and interface design. (but see Sardar, 2000)

We now understand some of the political implications of software design (Bodker, 1991); i.e. how software design largely influences and structures the work patterns of information workers sitting in their offices all over the world. This has resulted in solutions such as 'participatory design' where the software designers collaborate with the people that will use the software in its design. However, these studies are mainly concerned with people working with productive tools such as spreadsheets or text editors or recreational software such as games. One thing is to create a survey as a software designer that focuses on office work-patterns or the logic of a gameplay, but another (and theoretically impossible) is to try to build a model of how artists want to work, what they want to express and by what means. This is precisely the area where artistic open source software situates itself, allowing users to define their own agendas and aesthetic values by resisting a closure in the design of the software environment.

## Conclusion: Creators vs. Consumers of Expressive Tools

The FLOSS tools mentioned above are practically all commentaries of the question of software consumption. As they are open source, they invite the artist-technologist to modify and transform the tool by collaborating in its design either privately or as part of the culture around the tool. The idea of collaborative artist-made operating systems, programming languages/frameworks and creative tools makes sense. They are the people that know and understand how and what is required in the process of making computational art. They are the people whose work involves the reinterpretation of the social ontology of our culture through re-categorisation. The situation where artists are forced to buy closed source software and thus constantly being pushed into ergonomic patterns or conceptual boxes by the companies that made them is now over. The 'software consumers' of the commercial world become 'software co-designers' in the FLOSS world – users who have the freedom to choose the form of their expression and the manner they express it.

## Bibliography

Margaret A. Boden, The Creative Mind: Myths and Mechanisms. London: Wiedenfield and Nicholson, 1990.

Susanne Bodker, Through the Interface: A Human Activity Approach to User Interface Design. Hillsdale, NJ: Erlbaum, 1991.

Gilles Deleuze & Félix Guattari, A Thousand Plateaus: Capitalism and Schizophrenia. London: Continuum, 1987.
Matthew Fuller, (ed). Software Studies: A Lexicon. Cambridge: MIT Press, 2008.

Casey Reas & Ben Fry, Processing: A Programming Handbook for Visual Designers and Artists. Cambridge: MIT Press, 2007.

Thor Magnusson & Enrike Hurtado Mendieta, 'The Acoustic, the Digital and the Body: A Survey on Musical Instruments', in Proceedings of the NIME 2007 Conference, New York, USA, 2007.

Aymeric Mansoux, Antonios Galanopoulos, & Chun Lee, 'pure:dyne' in Proceedings of the 2007 Pd Convention. Montreal, 2007.

James McCartney, 'Rethinking the Computer Music Language: SuperCollider' in Computer Music Journal, 26:4, pp. 61-68, Winter 2002. MIT Press.

Click Nilson, 'Live-Coding Practice' in Proceedings of the NIME 2007 Conference, New York, USA, 2007.

Miller Puckette, 'Using PD as Score Language' in Proceedings of ICMC 2002, 2002. pp. 184-187.

Ziauddin Sardar, 'ALT.CIVILIZATIONS.FAQ: Cyberspace as the darker side of the west' in The Cybercultures Reader, Bell, David & Kennedy, Barbara M. (eds.). London: Routledge, 2000.

Andrew Sorensen & Andrew Brown. 'aa-cell in Practice: An Approach to Musical Live Coding' in The Proceedings of ICMC 2007, Copenhagen, 2007.

Toplap, http://www.toplap.org - accessed 3 November, 2007.

Jaromil, http://www.dynebolic.org - accessed 3 November, 2007.

Linus Torvalds, 'Linux' in Ars Electronica, 2003, http://www.aec.at/en/archives/prix_archive/prix_projekt.asp?iProjectID=2183

Fernando Lopez-Lezcano, 'Surviving on Planet CCRMA, Two Years Later and Still Alive' in Proceedings of the 2005 Linux Audio Conference. Karlsruhe:ZKM, 2005.

## Notes

This text draws on a survey polling artist-technologists exploring the time/creativity question and interpret their answers into 13 traces of thinking about tool use. We look at the question of strata in production environments and the question of finding a platform where the focus is on the enabling of general production (the time factor) with as minimum limitations as possible. Finally, we discuss the status of open source artist-created software environments as meta-art, i.e. expressive platforms that necessarily embody aesthetic ideas and purpose.

## Acknowledgements

Many thanks to Anonymous, Marije Baalman, Ross Bencina, Tom Betts, Andrew Brown, Graham Coleman, Nick Collins, Alberto di Campo, John Eacott, Tom Hall, Enrike Hurtado-Mendieta, Aymeric Mansoux, Click Nilson, Julian Rohrhuber, Andrew Sorensen, Matthew Yee-King and Johannes M. Zmoelnig for their brilliant answers to my simple question on time and expressivity in open source production environments. The opinions in this article (outside of italic quotations) are my own and I apologise if I have misrepresented the contribution of the respondents.