Contemporary Music Review
Vol. 28, No. 1, February 2009, pp. 89–101

Tools to Fight Boredom: FLOSS and GNU/Linux for Artists Working in the Field of Generative Music and Software Art

Marloes de Valk

This article takes a look at the impact the operating system, programming languages and software, as a whole, have on the practice of artists working in the field of generative music and software art. Proprietary operating systems lack the openness needed to create an environment that fulfills the specific needs of artists and musicians who program and programmers who produce art and music. 'Hackability', the possibility to take things apart, modify, adjust, and improve, is an ever more important aspect that software artists and electronic musicians seek to include in their production environment. GNU/Linux and Free/Libre/Open Source Software (FLOSS) possess this feature, and many more, providing artists with a truly creative and open environment, free of unnecessary technical limitations, predetermined interaction, lack of control over the work environment and dependence on software companies.

Keywords: Software Art; Generative Music; Free/Libre/Open Source Software; Artistic Practice; Artistic Toolkit; GNU/Linux

Introduction

The software toolbox for artists working in the field of generative music and software art often consists solely of the operating system (OS) and a programming language, sometimes completed with some software applications. This environment can be considered their instrument. In music as well as fine art, an artist's instruments or tools are carefully selected. The same care is needed in the selection of tools in digital art practice. It is very important to see how the operating system, when combined with a programming language and software

ISSN 0749-4467 (print)/ISSN 1477-2256 (online) © 2009 Marloes de Valk
DOI: 10.1080/07494460802664056

applications, impacts on an artist's practice. This article takes a practical look at the particular needs of artists working in the field of generative music and software art, and how Free/Libre/Open Source Software (FLOSS) and GNU/Linux can influence and redefine the relationship between creative process and artistic output. The second part of the article addresses how the use of GNU/Linux influences the shelf life of a work, and how it *could* mess up your artistic career by seducing you to stop making art; but probably won't. The article finishes with a note on the lack of documentation that many FLOSS projects suffer, and the efforts made to solve this.

> Many books, and many teachers, try to separate computer languages from the environment in which they run. That's like talking about cooking an egg without indicating whether the egg will be cooked in a microwave oven, fried in a frying pan, or boiled in a pot of water. The environment affects what you can do and how you can do it. (Burtch, 2004, p. 1)

When talking about artists and musicians here, I'm referring to people who create music, audiovisual performances or multimodal installations using generative techniques, which, in practice, often means using programming languages. The term FLOSS is used to refer to Open Source applications used in combination with an artist's own code. Discussing the use of higher level artistic software is outside the scope of this article, mostly because this type of software is hardly ever used in a generative context, which requires working in a bottom-up way, working on and experimenting with algorithms, instead of in a top-down way, using ready-made procedures. I'm presuming a DIY attitude, a way of working that includes both concept development and technical implementation of ideas, all done by the artist him or herself, not by hiring a programmer. Regardless of some preconceptions, artists and musicians do program and programmers can become effective digital artists. When working with generative techniques creatively, artists need to understand their medium, and there is only one way to gain true understanding: getting your hands dirty.

I'll be discussing the advantages and pitfalls of working with FLOSS and GNU/Linux, based on my own experiences as an artist working in the field of multimodal installations and audiovisual performances. I started experimenting with generative techniques using proprietary software, and slowly moved to a Free Software – based art practice; I experienced the impact this switch in environment had on my practice and witnessed similar experiences when working with colleagues. I'm writing this article as a plea for a way of working facilitated by FLOSS and more specifically by GNU/Linux: a modular, creative, and free use of the operating system which enriches the works of art created and provides artists with the freedom and flexibility needed in their practice. I will also open a window of speculation on the future of audiovisual tools and data released under free licenses and how this could provide a solution to software decay and distributed archiving.

Essential Items in an Artist's Toolbox

What specific needs do artists working in the field of generative music and software art have when it comes to their operating system, programming language and software? The most important one is the need for freedom to express concepts without being hindered by unnecessary technical limitations. Next to that, the environment should provide sufficiently complex interaction. This creates a wide range of expressive possibilities, enabling the artist to achieve a certain virtuosity in using it as an instrument. There is a need for control over the tools used in this process, requiring a reliable environment and independence for the artist. There is a need for tools that match an artist's needs, instead of tools that match what an industry determines an artist needs. Last but not least, an artist's toolbox should contain copyleft[1] licenses that can be applied to the work created. This opens up the otherwise opaque world of software and aims for the sharing and exchange of code, building on existing knowledge instead of reinventing the wheel.

No Unnecessary Technical Limitations

When materializing a concept, the limitations in this process should be determined by the choice of medium. A painter for instance, chooses to work within the range of possibilities inherent in the medium paint. He'll accept that paint, once dry, can't produce sound, but he will not accept unnecessary limitations, such as paint that doesn't stick to a canvas. In a medium such as paint, it is simple to determine which limitations belong to the medium, and which are unnecessary. When thinking about a technically more complex medium, such as software, it becomes much more difficult to make this distinction. On top of that, in the case of generative music and software art, software can be both medium and tool, and this certainly doesn't simplify things. Let's look at a few practical examples to clarify what this freedom means in the context of generative music and software art.

When programming your own generative systems for the creation of sound or image, having access to lower-level parts of the operating system and software is a big advantage. It allows you to customize the environment to fit your unique requirements. There is no technical reason why you wouldn't be able to have access to every aspect of the operating system and the software running on it (it helps if there is a 'safety switch', so unintentional modifications are ruled out). Access to the core of the operating system, being able to modify and customize it, for instance, would allow you to tweak your system to have better low-latency performance, something essential when performing live, or in interactive sound installations. If you are working on a heavy application and need to economize on CPU cycles, you can strip your system of all unnecessary bloat, leaving only the bare essentials. On top of that, access to the source code of the software, facilitating the possibility of compiling it with optimizations for specific hardware or purposes, can mean a big improvement in performance as well—never mind the possibility of actually modifying the source code of the software itself to match your needs. All these advantages can be summed up in one word as hackability, which is one of the main advantages of using GNU/ Linux.

Another feature of software as a medium that is often unnecessarily blocked is the freedom to mix and match different applications without bumping into incompatibility issues, making it possible to use system and software as a creative toolbox in which all parts can be used together and in different configurations. GNU/Linux in combination with FLOSS is an environment in which anything can be glued to anything using simple scripts, standard formats, and clever tools from JACK[2] to shell scripting with Bash.[3] JACK, for example, is a low-latency audio server, designed to connect different audio applications to each other and to an audio device (i.e. a soundcard). Previously, Linux audio applications worked with OSS,[4] then ALSA[5] came along, but nowadays the minimum requirement for audio software running on Linux is having JACK support. JACK functions as a glue between applications and greatly enhances the creative possibilities for artists. It also has benefits for developers of audio applications, since including JACK support means a larger user base. Shell scripting enhances the system in an even more profound way, since it allows you to script the behavior of the whole operating system. MS Batch files, Apple script and shell script in OSX are similar, but are either not as powerful or not as well integrated into the operating system as shell scripting with Bash is in GNU/Linux. Shell script is a very powerful tool that makes the operating system completely customizable and adaptable to your wishes.

A concrete example of this can be found in the development of the installation 'hello process!' by Aymeric Mansoux and myself. The installation consists solely of a computer and a printer, where the computer functions as it usually does, as a black-box theatre of processes. The only output comes through the printer, giving clues about the activity inside, while in the background, the raw noise of the machine creates a soundscape, a sonification of this theatre of naïve computation. The installation makes use of an old dot matrix printer for the output, and the main processes are programmed in Gforth.[6] What we wanted was to not send text or images to the printer, but to send numbers that would correspond to graphic elements from the printer's graphics mode, so we could generate abstract patterns that would create a map of the processes taking place inside the computer (see Figure 1). In order to do this, we needed direct access to the printer's graphics mode, so we wrote a shell script that directly interpreted numbers and turned them into the right escape sequence for the printer. Shell script was also used to glue all components of the installation together.

This freedom to mix and match is often blocked on proprietary systems using proprietary software, mainly because of the lack of standards. Commercial software often uses closed formats in order to get a competitive edge. Once users start working with a software that has a certain format as output, they enter a feedback loop of wanting to stay with this software in order to be able to keep working with old data, and because other people use this format too (to read your data and exchange files). If every company would work towards interoperability and open standards, people

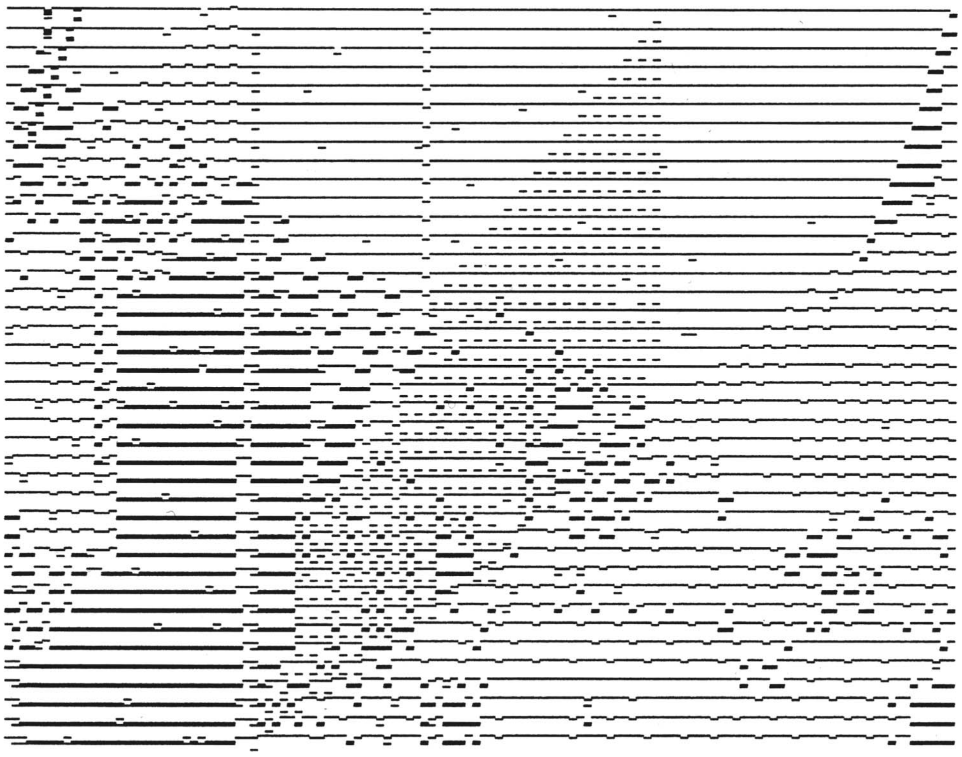


Figure 1 Print from the installation 'hello process!' produced by Mansoux and de Valk in 2008.

could use a combination of different vendor-specific software that suits their needs and workflow better, without having to deal with difficult exchanges and backward compatibility. Due to closed formats and incompatibility between different software, you are often forced to stay within the boundaries of one closed format, one collection of software from the same vendor, or bend over backwards to find workarounds to incompatibility issues.

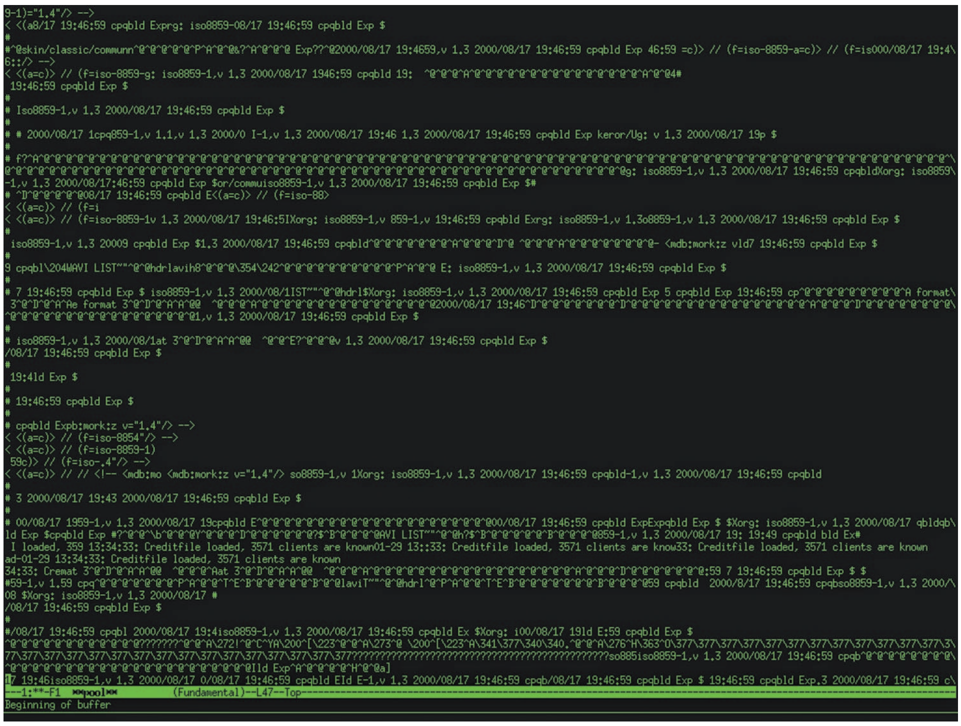
Unconventional Approaches

Traditionally, artists and musicians have always looked for ways to expand the expressive possibilities of a medium, both technically and conceptually. These experimentations and new approaches to a medium require the freedom to dissect, hack and conceptually take apart work done in that medium. The environment that an artist works in needs to be expansible and flexible. When working with programming languages and an operating system, this means an open environment where the artist is free to take unconventional paths. Small, self-contained environments don't allow for alternative approaches. They simply don't posses sufficient complexity in the way you can interact with them. The environment an artist uses needs to have the same kind of complexity in the range of possible interactions with it that acoustic instruments have, in order to allow for the construction of many non-repetitive creative systems. This doesn't mean a tool or environment needs to be incredibly complex itself, it means that it must provide enough freedom to allow the construction of instruments and artistic sandboxes. It is this complexity that gives you the freedom needed to experiment and innovate, that is the base for technical and conceptual virtuosity.

A good example of this experimentation and pushing of boundaries, making full use of the open and complex nature of GNU/Linux, is Martin Howse's 'PromiscuOS', a non-functional artistic operating system. The project aims to modify and redistribute an already existing GNU/Linux OS, in which different changes operate to remove security, segmentation and objectification from all levels of the system in favour of the generation, distribution and execution of code across all of its instances (see Figure 2).

Control, Reliability and Independence

NASA, the outfit that rockets people off into outer space for a living, has an expression: 'Software is not software without source code.' (Young, 1999)



```
9-1)="1.4"/> -->
< <(a8/17 19:46:59 cpqbld Exprg: iso8859-08/17 19:46:59 cpqbld Exp $
#

#^@skin/classic/communn^@^@^@^@^@^P^A^@^@&?^A^@^@^@ Exp??^@2000/08/17 19:4659,v 1.3 2000/08/17 19:46:59 cpqbld Exp 46:59 =c)> // (f=iso-8859-a=c)> // (f=is000/08/17 19:4\
6::/> -->
< <(a=c)> // (f=iso-8859-g: iso8859-1,v 1.3 2000/08/17 1946:59 cpqbld 19:  ^@^@^@^A^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^A^@^@4#
 19:46:59 cpqbld Exp $
#

# Iso8859-1,v 1.3 2000/08/17 19:46:59 cpqbld Exp $
#

# # 2000/08/17 1cpq859-1,v 1.1,v 1.3 2000/0 I-1,v 1.3 2000/08/17 19:46 1.3 2000/08/17 19:46:59 cpqbld Exp keror/Ug: v 1.3 2000/08/17 19p $
#

# f?^A^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^\
@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@g: iso8859-1,v 1.3 2000/08/17 19:46:59 cpqbldXorg: iso8859\
-1,v 1.3 2000/08/17:46:59 cpqbld Exp $or/commuiso8859-1,v 1.3 2000/08/17 19:46:59 cpqbld Exp $#
# ^D^@^@^@^@^@08/17 19:46:59 cpqbld E<(a=c)> // (f=iso-88>
< <(a=c)> // (f=i
< <(a=c)> // (f=iso-8859-1v 1.3 2000/08/17 19:46:5IXorg: iso8859-1,v 859-1,v 19:46:59 cpqbld Exrg: iso8859-1,v 1.3o8859-1,v 1.3 2000/08/17 19:46:59 cpqbld Exp $
#

 iso8859-1,v 1.3 20009 cpqbld Exp $1.3 2000/08/17 19:46:59 cpqbld^@^@^@^@^@^@^@^A^@^@^@^D^@ ^@^@^@^A^@^@^@^@^@^@^@^@- <mdb:mork:z vld7 19:46:59 cpqbld Exp $
#

9 cpqbl\204WAVI LIST~"^@^@hdrlavih8^@^@^@\354\242^@^@^@^@^@^@^@^@^@^@^P^A^@^@ E: iso8859-1,v 1.3 2000/08/17 19:46:59 cpqbld Exp $
#

# 7 19:46:59 cpqbld Exp $ iso8859-1,v 1.3 2000/08/1IST~"^@^@hdrl$Xorg: iso8859-1,v 1.3 2000/08/17 19:46:59 cpqbld Exp 5 cpqbld Exp 19:46:59 cp^@^@^@^@^@^@^@^@^A format\
 3^@^D^@^A^Ae format 3^@^D^@^A^A^@@  ^@^@^@^A^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@2000/08/17 19:46^D^@^@^@^@^@^@^@^D^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^A^@^@^@^D^@^@^@^@^@^@^@^@\
^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@1,v 1.3 2000/08/17 19:46:59 cpqbld Exp $
#

# iso8859-1,v 1.3 2000/08/1at 3^@^D^@^A^A^@@  ^@^@^E?^@^@^@v 1.3 2000/08/17 19:46:59 cpqbld Exp $
/08/17 19:46:59 cpqbld Exp $
#

 19:41d Exp $
#

# 19:46:59 cpqbld Exp $
#

# cpqbld Expb:mork:z v="1.4"/> -->
< <(a=c)> // (f=iso-8854"/> -->
< <(a=c)> // (f=iso-8859-1)
 59c)> // (f=iso-.4"/> -->
< <(a=c)> // // <!-- <mdb:mo <mdb:mork:z v="1.4"/> so8859-1,v 1Xorg: iso8859-1,v 1.3 2000/08/17 19:46:59 cpqbld-1,v 1.3 2000/08/17 19:46:59 cpqbld
#

# 3 2000/08/17 19:43 2000/08/17 19:46:59 cpqbld Exp $
#

# 00/08/17 1959-1,v 1.3 2000/08/17 19cpqbld E^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@00/08/17 19:46:59 cpqbld ExpExpqbld Exp $ $Xorg: iso8859-1,v 1.3 2000/08/17 qbldqb\
ld Exp $cpqbld Exp #?^@^@^\b^@^@^@Y^@^@^@^D^@^@^@^@^@^@?$^B^@^@^@^@AVI LIST~"^@^@h?$^B^@^@^@^@^@^B^@^@^@^@859-1,v 1.3 2000/08/17 19: 19:49 cpqbld bld Ex#
 I loaded, 359 13:34:33: Creditfile loaded, 3571 clients are known01-29 13::33: Creditfile loaded, 3571 clients are know33: Creditfile loaded, 3571 clients are known
ad-01-29 13:34:33: Creditfile loaded, 3571 clients are known
34:33: Cremat 3^@^D^@^A^A^@@  ^@^@^@^Aat 3^@^D^@^A^A^@@  ^@^@^@^A^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^A^@^@^@^D^@^@^@^@^@^@^@:59 7 19:46:59 cpqbld Exp $ $
#59-1,v 1.59 cpq^@^@^@^@^@^@^@^@^P^A^@^@^T^E^B^@^@^@^@^@^@^B^@^@laviT~"^@^@hdrl^@^P^A^@^@^T^E^B^@^@^@^@^@^@^@^@^B^@^@^@^@59 cpqbld  2000/8/17 19:46:59 cpqbso8859-1,v 1.3 2000/\
08 $Xorg: iso8859-1,v 1.3 2000/08/17 #
/08/17 19:46:59 cpqbld Exp $
#

#/08/17 19:46:59 cpqbl 2000/08/17 19:4iso8859-1,v 1.3 2000/08/17 19:46:59 cpqbld Ex $Xorg: i00/08/17 19ld E:59 cpqbld Exp $
^@^@^@^@^@^@^@^@^@^@^@^@???????^@^@^A\272!^@^C^YA\200^[\223^@^@^A\273^@ \200^[\223^A\341\377\340\340.^@^@^A\276^H\363^O\377\377\377\377\377\377\377\377\377\377\377\377\3\
77\377\377\377\377\377\377\377\377\377\377\377\377\377\377??????????????????????????????????????????????????so885iso8859-1,v 1.3 2000/08/17 19:46:59 cpqb^@^@^@^@^@^@^@^@\
^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@Ild Exp^A^@^@^@^@^H^@^@a]
17 19:46iso8859-1,v 1.3 2000/08/17 0/08/17 19:46:59 cpqbld EId E-1,v 1.3 2000/08/17 19:46:59 cpqb/08/17 19:46:59 cpqbld Exp $ 19:46:59 cpqbld Exp.3 2000/08/17 19:46:59 c\
---1:**-F1  **pool**        (Fundamental)--L47--Top-----------------------------------------------------------------------------------------------------------------------
Beginning of buffer
```

Figure 2 Screenshot from the work 'promicuOS' produced by Martin Howse in 2005.

NASA cannot rely on software unless it has control over every line of code in it, and although musicians and artists don't usually shoot billions of dollars into space, they do need control over the tools they work with. I find it very hard, not to mention extremely frustrating, to rely on a company for bug fixes and the implementation of new features. In the world of Free Software I might have to wait as well, but the feedback loop between users and developers is much tighter compared with the proprietary system, and anyone can provide a fix or a new feature. If you're handy, you can even do it yourself. This is not only about control over your tools, it is about independence. When you own the system and applications you use, instead of only having a temporary license to use them under certain terms, you can do what you want with them. You can modify, copy and redistribute without breaking any End-User License Agreements. The chances of running into the need for new features in the first place are a lot greater when using proprietary software. The market is flooded with applications that match what the software industry 'thinks' artists need, clouded by marketing strategies and target audience confusion that come with top-down development methods (O'Reilly, 1999, p. 195). In the open source model of development, the user and developer are much closer to each other, resulting in software that fulfills real needs as opposed to presumed ones.

This independence to do things yourself, and the possibility of studying and modifying existing source code leads to many new features, fixes and even small new applications. Musician Karsten Gebbert wrote a small commandline application that extended the functionality of the tools he uses to create and record music. He was looking for a way to sync a MIDI drum computer's output to the recording timeline of the digital audio workstation software Ardour[7] to facilitate editing the recorded material. He bumped into a limitation of Ardour: it only produces MIDI Time Code (MTC) as synchronization clock source. MTC is absolute time rather than musical time (MIDI Beat Clock), and therefore did not fit his purpose. He started to learn C and studied the source code of the JACK API to write a little commandline application that produces MIDI Beat Clock (and MTC) synced to the global JACK transport, sending it to JACK's MIDI ports (Karsten Gebbert, personal communication, July 2008).

Digital Alchemy versus Sharing and Exchange of Code

> But why should people who program computers be so concerned about copyrights, of all things? Partly because some companies use *mechanisms* to prevent copying. Show any hacker a lock and his first thought is how to pick it. But there is a deeper reason that hackers are alarmed by measures like copyrights and patents. They see increasingly aggressive measures to protect 'intellectual property' as a threat to the intellectual freedom they need to do their job. And they are right. It is by poking about inside current technology that hackers get ideas for the next generation. (Graham, 2004, p. 51)

When, in addition to using Free Software and an open source OS, artists use copyleft licenses to publish or release their work, the atmosphere of digital alchemy and secret algorithms can be replaced with a more constructive one. Instead of reinventing the wheel, we can build upon existing knowledge and techniques to enrich and progress. Copyleft can play an important role in this demystification of software art and generative music. Another advantage of using copyleft is peer review of the code you write. This is a strong motivation to produce good code and a great way to improve its quality in the process. Perhaps the most obvious benefit of copyleft is that it makes sharing and reuse of code possible. This has great educational value but also practical advantages because it can greatly reduce the time needed to put together an application when combining existing parts into something new.

> That ideas should freely spread from one to another over the globe, for the moral and mutual instruction of man, and improvement of his condition, seems to have been peculiarly and benevolently designed by nature, when she made them, like fire, expansible over all space, without lessening their density in any point, and like the air in which we breathe, move, and have our physical being, incapable of confinement or exclusive appropriation. Inventions then cannot, in nature, be a subject of property. (Jefferson, 1813)

Next to these practical arguments, there is the issue of pure and simple economics. The economic model that has been applied in the arts is going to become obsolete rather soon. More artists using copyleft will contribute a great deal to bringing parts of the art world into the twenty-first century, finally getting rid of outdated ideas and models. One of the most persistent of these ideas is the concept of the genius artist, who somehow has a unique insight or idea that no one else has had before her, and which is her possession. Secrecy surrounding the code or the production process behind a work of art or piece of music is kept, perhaps for this reason, but probably mostly because it is still somehow thought to be the only guarantee to making a living as an artist. It's not really the artist or musician who is to blame for the survival of this idea, it's the art market and the music industry that haven't moved on for decades. Let's look at the art market first. The value of a work of art is directly linked to its uniqueness and scarcity. This was not a big problem for photography, video or installation art, because these can all be made into 'limited editions', artificially creating uniqueness. But more recent forms of art such as software and net art don't fit this model: there is no scarcity, for they are as infinite as the ideas in Jefferson's quote. The music industry has also taken the road of artificial scarcity, using copyright to prevent their product from becoming infinite. It seems art and music will finally head in the same direction, the one in which economic value is increased by giving the non-scarce parts of a work of art or piece of music away for free. The real value is in the scarce parts, the artist or musician's presence and performance, their knowledge, and their next project (Masnick, 2007).

Shelf Life

An important factor in the choice by an artist about the environment he will produce his work in is the effect this choice has on the shelf life of his work. In the early days of software art and generative music there wasn't much attention paid to this aspect, which resulted in a couple of generations of artworks being lost for ever because they are now impossible to run. Since then, a lot more awareness has arisen, but it's still a difficult issue. Once you've created a work using a certain software and/or a certain output format, your work can enter the world and circulate. But as time goes by, formats change, software is updated or discontinued, and new hardware appears. How can you make sure your work can still be experienced years after you finished developing it?

In general, the use of proprietary operating systems and software has a negative impact on the shelf life of a work of art. There are several reasons for this. The chance that software, versions of operating systems and formats might be discontinued or no longer supported is great. It's not commercially advantageous to put a lot of effort into maintaining and supporting older versions that run on older hardware. When a user decides to stick with a software or system but updates to a newer version, there is quite a significant possibility that these newer versions won't work with older hardware. Last but not least, in some cases your license of the software or your rights over digital data will end at some point. After that you're not allowed to use it anymore. And if you buy a new license, it will be for a new version of the software (if it still exists) and this version might not be compatible with your work.

FLOSS, on the other hand, allows you to keep your copy of the OS and software for ever. So as long as you make sure the OS can still run on some hardware or other, you need to only store that particular version of the OS and that particular version of the software to be ensured your work can still be run or played. Backward compatibility is also much better in GNU/Linux and FLOSS. With these systems there is no commercial imperative to only support the latest hardware; older hardware is used by many and therefore things are maintained much longer. Because FLOSS is based on standards instead of closed formats, and backwards compatibility gets a lot of attention, the chances that your work will still run in the future are a lot better. Even if there are problems running your application, you have access to the source code of your work, the system it used to run on and the system you are trying to run it on. You can still modify the software or newer versions of it to fit your needs. If you use Free Software and an open OS, but can't get hold of the original hardware, it is easier to emulate the operating system and code, because you have the source of both system and application. You can port the work directly to another platform if need be, instead of having to reverse engineer the application, starting with a binary and an idea of which hardware it was running on. This is one of the problems faced by art archives and galleries if they only have possession of a binary file made for an old OS. Unless this fleeting quality is what you're after, longevity is an important factor in the choice of an environment to produce your work in.

Negotiation between Concept and Tool

How do you prevent getting lost in endless technical experimentations when the possibilities within GNU/Linux and FLOSS are so vast? Just like any other tool that allows for complex interaction, there is always the risk of the technical side becoming dominant and the artistic expression of concept being lost or buried under technology. This is the flip side of the freedom that you have in this environment. It's both an advantage and a risk. More restrictive environments will not allow you to stray from their partly predetermined path. The artist must find a balance between technical implementation of a concept and the artistic expression of it.

Next to this conceptual balance, there is the issue of time investment. Because of the freedom within the GNU/Linux environment, you can easily end up investing a lot more time into the development of a project than with more restrictive environments. You are more likely to go for non-standard solutions, choose to program things yourself instead of relying on readymade solutions, and so on. When an artist is new to Linux and FLOSS, an initial investment in the technical side is needed. It simply takes time to learn how to work within this new environment. After that, the artist should be able to implement their ideas just as quickly as with any other tool. It's both a matter of finding a balance and a matter of setting a goal. If the goal is to work towards a finished product, the issue of time investment is important. If the goal is to develop an instrument, and this development process is more important than any end product or even the use of it, the time investment becomes much more justifiable.

Because Linux and FLOSS blur the line between developer and user, the risk of ending up developing tools instead of works of art is a lot greater. There is nothing wrong with developing new tools, but if your goal was to create an audiovisual performance and you end up with an incredible new application but no idea what to do with it on stage, you might feel cheated. It sounds like a simple thing to avoid, but in practice, when programming your own applications in an open development environment, it is all too easy to forget about having to perform and instead spend all your time on building an amazing instrument. More and more artists now blur the distinction between the development of the tool and the artwork, as shown in a recent survey amongst artist/technologists (Magnusson & Hurtado Mendieta, 2007):

> 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 . . . (Magnusson, 2008)

Using GNU/Linux and FLOSS in an artistic practice brings an unavoidable political message to the work. When the use of FLOSS is advocated more prominently than the artistic concept itself, a work of art will not be understood on the right terms.

The political message that all work made with FLOSS carries in it should not be in the foreground, unless this is part of the artistic concept. The artwork itself, its quality and its inspiration should be the focus point—just as ethically traded goods are expected to be of good quality first and foremost if they are ever to be successful in the marketplace. You cannot compete artistically on the basis of ethics alone, as demonstrated by the success of unethically sourced but cheap products. Besides—and this is worse—the audience will completely miss the point of your work. But the political dimension of using Free Software is an important one, placing the artistic work in a bigger context. It implies an awareness of the fact that software can be the base material of a work of art or a piece of music, like the clay of a sculpture, and that this choice of material greatly influences the eventual work and its cultural context (Mansoux & de Valk, 2008).

Documentation

One big issue often mentioned when it comes to FLOSS and GNU/Linux is the lack of documentation. The writing of documentation is often at the very bottom of the priority list for developers. And if there is documentation, it can be quite a struggle to figure out how things work, since it is often written by developers and they tend to have a very special style for explaining things, often not understandable by mere mortals. This is a problem that is getting more and more attention, but is far from being solved. Luckily, there are a lot of users who have organized themselves into online communities, providing each other with much-needed help. There are countless mailing lists, forums, and IRC channels where assistance can be found. The more specialized and obscure the topic, the harder it will be to find good documentation. Artists working in the field of generative music and software art will often find their interests lie in the least documented parts of the world of software and code.

There are several efforts made to improve the accessibility of GNU/Linux and FLOSS by improving the documentation surrounding it. The FLOSS manuals project,[8] by Adam Hyde, for example, is a project that aims to set up-easy-to-read manuals for artists, by artists, avoiding the specialized lingo of developers. The *Digital Artists' Handbook*,[9] produced by folly,[10] is a valuable source of information that introduces artists to different tools, resources and ways of working related to digital art and FLOSS. All articles are written by experts in their field. Another project worth mentioning is pure:dyne,[11] a GNU/Linux live distribution for media artists, that offers a complete set of tools for real-time audio and video processing. This project makes working with Linux and FLOSS more accessible and it also provides a growing resource of easy-to-read documentation on the OS itself and some of its applications. Though very helpful, most of these projects are targeted at new users, not experts. If an artist or musician ends up struggling to make a piece of code work, written in a not well-documented language, the only way out is often still a huge amount of persistence and creativity.

Conclusion

We shape our tools and thereafter our tools shape us. (McLuhan, 1964)

Once you're used to working with a certain OS, programming language or software, you often grow to love it and become quite attached to it. Not because it is the best environment for you to work in, but because you can fluently express yourself in it. You know all solutions the tool has to offer and can formulate your problems within this range of solutions. As Maslow points out in *The Psychology of Science* (1966), if the only tool you have is a hammer, it is tempting to treat everything as if it were a nail. When trying out another environment you feel handicapped, it's very frustrating, nothing comes naturally, and you feel as if your hands have been chopped off. Returning to your own trusted tools afterwards gives you the same kind of joy as speaking in your mother tongue after having spoken a second language for a while. But this should not stop you from critically examining the tools you work with from time to time. And sometimes it is worth switching, no matter how painful this may be at first. Since the impact of the tools you work with on your practice is so great, and touches upon so many aspects of it, not just the technical ones, all efforts will pay off.

> Our musical alphabet is poor and illogical. Music, which should pulsate with life, needs new means of expression, and science alone can infuse it with youthful vigor. Why, Italian Futurists, have you slavishly reproduced only what is commonplace and boring in the bustle of our daily lives? I dream of instruments obedient to my thought and which with their contribution of a whole new world of unsuspected sounds, will lend themselves to the exigencies of my inner rhythm. (Varese, 1917, quoted in Kostelanetz & Darby, 1996)

In the end, the most desirable work environment is the one that provides the fewest obstacles when transforming ideas and thoughts into visual or audible reality. A great deal of inventiveness is needed to create such an environment. In order to not, as Varese states, slavishly reproduce what is already there, artists need to take things apart, modify, adjust, improve, break, and hack. GNU/Linux, FLOSS and copyleft all contribute to making such an environment possible, and on top of that stimulate the Free Software movement, the freedom of culture and art, and the independence of artists to create, share and publish.

Notes

[1] Copyleft is a form of licensing that uses copyright law to remove restrictions on distributing copies and modified versions of a work and requires that the same freedoms be preserved in modified versions. The most famous example of a copyleft license is the GNU General Public License.

[2] JACK audio connection kit.

[3] Bash is a Linux shell written for the GNU project and is the standard shell in most Linux distributions. Its name is an acronym taken from 'Bourne again shell', a reference to the first Unix shell, written by Steven R. Bourne.

[4] Open Sound System (OSS) was the first attempt to unify the digital audio architecture for UNIX and UNIX-compatible operating systems.

[5] The Advanced Linux Sound Architecture (ALSA) followed OSS, and provides device drivers for soundcards and MIDI functionality to the Linux operating system.

[6] Gforth is an implementation of the ANS Forth programming language, developed within the GNU project. Forth is a procedural, stack-based language, combining a compiler with an interactive shell.

[7] Ardour is a digital audio workstation that can be used to record, edit and mix multi-track audio at a professional level.

[8] http://en.flossmanuals.net.

[9] http://digitalartistshandbook.org.

[10] folly is a digital arts organization developing and delivering a programme of online work, live events, presentations, learning, research and consultancy work.

[11] http://puredyne.goto10.org.

References

Burtch, K. O. (2004). *Linux shell scripting with Bash: A comprehensive guide and reference for Linux users and administrators* (1st ed.). Indianapolis: Sams Publishing.

Graham, P. (2004). *Hackers and painters: Big ideas from the computer age.* Sebastopol, CA: O'Reilly.

Jefferson, T. (1813). Letter to Isaac McPherson. In A. A. Lipscomb & A. E. Bergh (Eds.), *The writings of Thomas Jefferson* (vol. 3, article 1, section 8, clause 8, document 12). Washington, DC: Thomas Jefferson Memorial Association.

Kostelanetz, R. & Darby, J. (1996). *Classic essays on twentieth-century music: A continuing symposium.* Florence: Wadsworth Publishing.

Magnusson, T. (2008). Expression and time: The question of strata and time management in creative practices using technology. In A. Mansoux & M. de Valk (Eds.), *FLOSS+Art.* London: Openmute.

Magnusson, T. & Hurtado Mendieta, E. (2007). The acoustic, the digital and the body: A survey on musical instruments. In *Proceedings of the 7th international Conference on New Interfaces for Musical Expression.* New York: ACM.

Mansoux, A. & de Valk, M. (2008). *Introduction*. In A. Mansoux & M. de Valk (Eds.), *FLOSS + Art.* London: Openmute.

Maslow, A. H. (1966). *The psychology of science: A reconnaissance.* New York: Harper & Row.

Masnick, M. (2007, 3 May). The grand unified theory on the economics of free. http://techdirt.com/articles/20070503/012939.shtml.

McLuhan, M. (1964). *Understanding media: The extensions of man.* New York: McGraw Hill.

O'Reilly, T. (1999). Hardware, software, and infoware. In C. DiBona, S. Ockman & M. Stone (Eds.), *Open sources: Voices from the open source revolution.* Sebastopol, CA: O'Reilly.

Young, R. (1999). Giving it away: How Red Hat Software stumbled across a new economic model and helped improve an industry. In C. DiBona, S. Ockman & M. Stone (Eds.), *Open sources: Voices from the open source revolution* (pp. 120–124). Sebastopol, CA: O'Reilly.