POETIC SOFTWARE

How artistic methods elicit critical reflection on software

INTRODUCTION

Software has taken command of our daily life.[1] It is omnipresent and most of our Western society would come to a halt without it. At the same time, software has become so ordinary, that it is often overlooked. Software is taken for granted while being increasingly entangled in our life and continuously adopting new tasks. Our computers seem to become smarter through new kinds of algorithms. This leads to new challenges in understanding software – not only from a scientific point of view but also from a cultural, political and social perspective. Software has also found its way into the art and vice versa, but there are still gaps in the relation between the two. I assume that the interaction between software and art can be productive and helpful for the research in both of the disciplines.

The question that I am asking is: How can artistic methods be used to elicit critical reflection on software as a cultural object beyond the interface? The current perception and use of software are significant parts of this research, especially in contrast to the original culture around software, which included hacking and required every artist to write their own software. This essay explores the multiple layers of software with a particular focus on the dependencies and imaginations that arise around and through software. How are the entangled, hidden layers of software coming to the surface?

Software consists of several parts, that could be divided into the code, the compilation, the execution and its manifestation (e.g. visual output on displays, or the computer reacting to mouse clicks). The code is a well researched topic and there

1 Referring to the book of Lev Manovich, *Software takes command*

have been many works that use code and programming for artistic purposes. The manifestation of software execution is what people are in contact with the most. The visual outcome on the screen is what determines how users perceive software. However, outcomes of software can also be invisible to the user, like data transmission, web servers or software for infrastructures. What is visible is mostly not the software itself, but the result of its execution (a webpage, or trains going back and forth). Execution is the most abstract part, but at the same time it is the most crucial part of software. During runtime, machine code[2] turns into machine commands and physical current, resulting, for instance, in a change of pixel colors. This complex interplay, when the code turns into machine action, is in itself an act of poetic expression – an interpretation of the code through the machine, an in-between state with clearly defined rhythmic. The exact moments of these transitions are beyond human perception. A division of software would only simplify the complex inter-dependencies the different parts have. It is precisely these moments of transition, the in-between states, the dependencies, that this essay tries to emphasize.

THE POETRY IN SOFTWARE

I consider poetry as a reference to the emotional, subtle and artistic expression that software can have. This work is not about considering software code as poems or as literature. It points to the non-neutral and imaginative character that

2 The human readable code is transformed into machine code through compilation. It is a process of translation from human readable instructions to machine instructions. Only the machine code can be executed by the machine, so the part of compilation is crucial to the creation of software.

software already has and that can be used for further artistic engagement. It also embraces the potential non-functional attributes of software and acknowledges the metaphors that software uses. It reflects on the different layers of interpretation and execution that software can have, and leaves the result open for interpretation.

Poetic software provides the possibility to create new artistic software, that is beyond the interface and beyond the expected mode of operation or depiction of software. Poetic software does not need to function but comes with an inherent call for statements about issues of software.

During the research for this project, I found myself returning to the essay *There is no Software* by Kittler over and over again, drawing inspiration from and following up on the various issues touched upon by Kittler. I uncovered a great variety of controversies surrounding the creation, execution and use of software. Furthermore, I realized that the more research I did on software and its implications for our lives, the more aware I became of the software I have been using. I started observing my attitude towards various applications that have been shaping my life and work every day, and started questioning many functions and backgrounds of software that I had viewed as a given before.

I became an ethnographer of my own work in progress. I realized that my behavior and everyday occurrences in the interaction with software reflected what I was reading in research papers and articles on my screen, and vice versa. Kittler serves as a point of departure for different controversies around software. This also leads me to the arts, and why I think art might provide possible approaches towards these different topics.

THE METHOD

The first part of my work is an ethnographically-inspired examination of the interaction with my computer, while reading Kittler's essay "There is no software". The text unfolds on two different levels: on the one hand, I am describing the process of reading while interacting with the software I use to do so. On the other hand, there are interventions to reflect on various concepts touched upon critically. These interventions refer to either Kittler's text itself, or to the software that I am using. In the second part, I am describing how art provides different frameworks to approach the different aspects I pointed out in the first part, and how art and software relate to each other's practices.

WHY THIS METHOD?

The detailed description of reading digitally makes the various software that is being used visible. Through that, the software can be observed while at work. Next to this, it is a great chance to revisit the text of Kittler. This method also allows for new encounters and associations with software, that help to recognize the different agents at stake when thinking about the processes of software and the involvement of art with it.

WHY 'THERE IS NO SOFTWARE' BY KITTLER?

This text very early became one focus of my interest and research. The text offers an excellent source for thinking about software today. Because in the essay from 1992 Kittler is not negating the existence of software, he instead wants to emphasize the materiality that is being neglected in his opinion. This is a huge tension that we can also recognize in computation today. Even if we do not dismiss the existence of software, it becomes more and more invisible. Workflows are so seamless, it seems almost like there is no software.

Fig. 1: The document viewer showing *There is no Software* by Kittler.

1. I AM READING, THE COMPUTER IS READING

or how to observe and understand the layers of software

I am reading *There is no software* by Friedrich Kittler. I downloaded the pdf file of the text onto my computer using the Firefox browser. The browser has saved the file in my downloads folder. I can find it through the file-system, which I can see in a representational view by opening the file explorer.

By clicking the icon of the file explorer, the computer opens a new window for me. I see different icons of folders and many other ones for files. I double click my way through the folders until I end up in the Downloads folder, where the newly downloaded file is placed in a list view among others. The file is called 'Kittler_Friedrich_1992_1997_There_Is_No_Software.pdf.' I hover over the small bar with the title. The operating system default setting is to open the file with the document viewer and so I do, by double-clicking the left mouse button. Within seconds, a new window appears, putting the file manager window into the background and foregrounding the title page of the pdf framed by small icons and scroll-bars. I click to enlarge to full-screen, and start to scroll down till the first lines of text appear. I zoom out pressing the combination CTRL and "-" twice. Next, I start reading the first sentence. "The present explosion of the signifying scene, which, as we know from Barry McGuire and A F. N. Dahran, coincides with the so-called Western world, is instead an implosion." Barry McGuire? I hover the name, press the mouse down and drag from "B" to "e". The text tints white with a blue background. The release of the mouse button is followed by pressing CTRL C. I switch to the browser, which still shows the download

page of the PDF. I paste the name into the search bar and press enter. The search engine shows a list of results – one video; this must be it. As the link reacts to my hovering, I click on it and, with a short flickering, I end up on YouTube. Without any action required, the video starts and the speakers play: "The eastern world it is exploding", to which Kittler must have referred.

→ COMPRESSION: TOO SMALL TO SEE, TOO LARGE TO FORGET

The implosion and explosion can well be seen on different levels of software. While the complexity and interplay of different technologies are exploding, the visibility and the potential for understanding are imploding. Increasingly better software brings great advances, for instance in computer vision, but at the same time, it becomes harder to understand. The potential of having more sophisticated technology may come at the risk of blurring the understanding. At the same time, these highly complex algorithms require more hardware and even better processors.

The implosion of files is a very well used method in the form of compression. Compression needs software that can rearrange the bytes of files using various algorithms, for the sake of file size. Smaller files can be stored easier and have advantages for transmitting. However, this can have different implications. It is a method to circumvent the physical limitations (to some extent). It means that files can be stored with very little storage available. Other than that, we produce increasingly bigger

files, because cameras output high-resolution images, we can gather more data, scan better and display highly sophisticated websites. How directly does this affect us? Unlike the imagination that the digital is immaterial, the processing of big files for instance is consuming much energy (De Decker, 2018). Therefore, some websites like lowtechmagazine.com are thinking about different methods on how to host low-energy web pages. They are using solar panels and produce their websites in a way that makes the site very light in terms of data that has to be transmitted. From this case we can see that compression can have multiple effects. It is the small nuances that make software a powerful tool to think about current cultural topics. This lightweight approach gives reason to think about different aspects of how websites are being served and how they are built.

Space is a recurrent scheme in computation. Computer Science tries to shrink and expand at the same time. It is almost like a play that can be observed on different layers. The 'compression of space' into the size of a microchip is opposed with the exploding need for power or, to remain within the metaphor of space, disk space (Kitchin, 2011). The expansion of the digital does not remain within the computer, but it is actively becoming part of our real space. "[S]oftware generates behaviors and opportunities, and traffics in meanings, readings, and interpretations" (Kitchin, 2011). To figure out these exact moments of influence and these borders between computers and the real world might

be very hard to accomplish, if not impossible. As Hu points out, the material and the digital world are interwoven more deeply than we think. For example, the imagination of the internet as a cloud manifests in the real world as cables that get placed across oceans, and buildings that hold thousands of servers (Hu, 2016). Therefore, it becomes clear that a separation between the digital and the material world does not make sense.

I stop the video by clicking onto the face of the singer and a smoothly appearing pause sign inside a circle signals the success of my action. I change back to the document viewer by clicking on the window that got hidden in the background by the browser.

The words I read are displayed with a grained border presumably caused by the scanning process. As I read on, my t-shaped cursor follows the lines of the text. I continue with the next sentence. "The last historical act of writing may well have been the moment when, in the early seventies, the Intel engineers laid out some dozen square meters of blueprint paper" (Kittler, 1992).

→ **I DEPEND ON YOU, WHO DO YOU DEPEND ON?**
With its increasing speed, computation fosters itself while depending on the previous version of its own. The same holds true for software. Therefore, we can recognize a spiral of dependencies and influences that includes humans and machines. After the first hardware was able to draw new, even smaller hardware than it would ever have been possible with paper and pen, the system of hardware

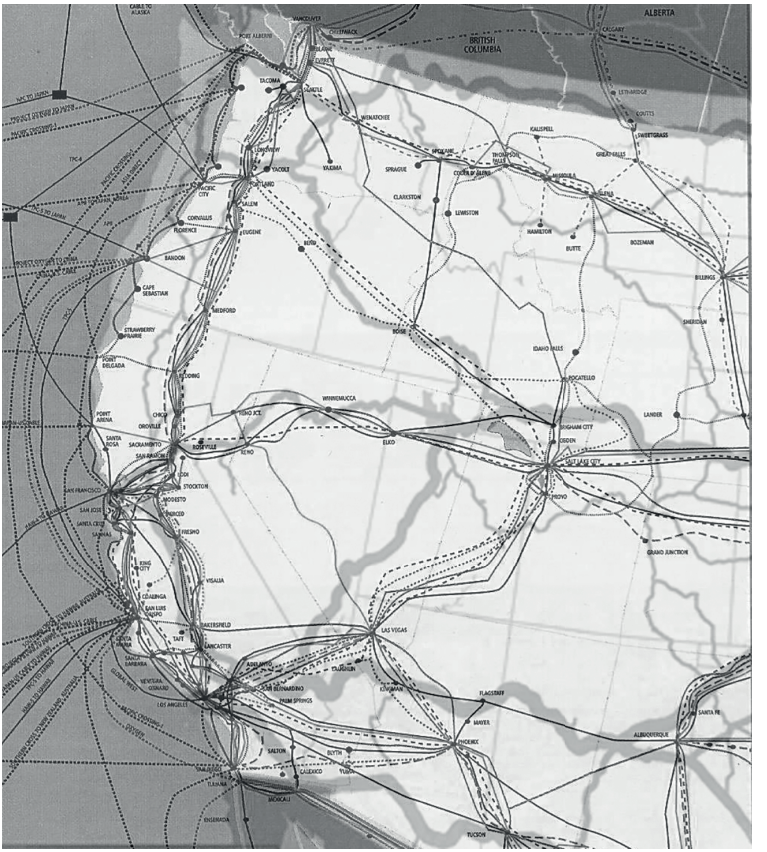

Fig. 2: The internet in its physicality.
Cables are following old railroad routes.

Fig. 3: The software of satellites manifests in the form of calibration targets in the desert.

design became dependent on itself (Kittler, 1992). This means that the next generation of hardware is always enabled by and relying on the previous version, making it possible to create even smaller and more complicated parts. The same can be found in the culture of software development. Software can only be built with software: software that enables to write the program code, software that compiles the code into machine-readable binary-code and an operating system that executes it. It means that nearly every program relies on other ones, requiring users to pre-install specific versions of software in order to run the program. If one single component of this chain of dependencies breaks, many other programs will be affected.

The dependence on companies that produce software is huge. If the company decides to discontinue their software, the users become immediately aware of their dependence, as they can not use the discontinued software anymore. This happened for instance when Microsoft shut down one of their scripting languages. Many companies that relied on it suffered (Ullman, 2012).

Software is changing over time. As Ellen Ullman mentions, this is also the reason why software can quickly become unstable, especially if multiple programmers are working on a program over a long period of time. Code can be written in many different ways, making it hard for other people to read or to understand. Still, huge systems with a long history have to be kept running as many other systems rely

on them resulting in a very fragile setup, in which you can hardly change anything (Ullman, 2012). These kinds of dependencies tell their own stories and are rarely clearly visible. These dependencies have the potential to show the history of software. It clearly shows that software cannot exist on its own, but is always embedded into a greater ecosystem, a cultural framework that follows its own rules.

"We shape our tools and, thereafter, our tools shape us," (Davis, 2016) says a famous quote by John Culkin from 1967. If we look at the dependencies of software, one could also say: we shape tools and these tools shape new tools. Transferring this idea to the notion of software as a cultural object, the interrelation between shaping and being shaped could be formulated as follows: software creates and influences culture, and therefore this culture shapes new social conditions under which the construction and use of software itself are altered. This might become clear when looking at the example of software hacking. The distribution of proprietary software with Digital Rights Management (DRM) leads to multiple groups cracking and circumventing software limitations. These cracks are then distributed as new software. The original culture of software was actually built around a free culture, that distributed code openly and freely. Early software production was very dependent on this openness (Mansoux, 2017). Without the sharing of software and code, the development would have been very tedious, if

possible at all. It is this openess that nowadays has to be defended, like the free software movement does. Free software is not self-evident anymore, because companies commercialized software for their profits.

I further follow the dark pixels on the screen to the roaring sound of the computer. It is not clear whether the ventilation sound is triggered by the hardware or the software, which is causing the CPU to overheat. Kittler is writing about how language gets abstracted from high-level, human-readable words, to assembler code, that is being translated into non-readable machine code. As Kittler talks about this "postmodern Tower of Babel" (Kittler, 1992, p. 148), I realize how my own windows have started to build up like a tower. The document viewer on top of the browser on top of the settings on top of the mail program, and so on.

→ **FRAMEWORK CULTURE**

Programming languages are based on other programming languages in order to make the code easier to write and read. Low-level languages are very close to the actual machine processes and therefore very complex to write. This is why high-level languages were constructed to translate these elaborate processes into human readable concepts and language. In addition to that, programmers often rely on third-party frameworks, which provide functions that are very convenient to implement. Instead of having to write the code themselves, they can just import it by using a single line of code. Therefore, the whole set of tools provided by the so-called library becomes available

for the use of the programmer. The process of using frameworks often obscures the actual algorithms. For example, it can be quite challenging to create a machine learning algorithm from scratch but frameworks like *Keras* or *Tensorflow* make it accessible. The problem is that the programming syntax is very close to human language, which makes the underlying code hard to grasp. Thus, it is harder to change functions that are underneath the layer of the framework-interface (Cox, 2007).

Furthermore, different programming languages favor different concepts of language and writing as well (ibid.). The choice of programming language already determines a certain style of writing. Because language significantly shapes our imagination, the choice of programming language also influences our understanding of software. Although scripting languages are very popular right now, they cannot replace low-level programming.

"High-level programming approaches can be very successful in achieving certain ends, but the very imposition of higher-level constructs and metaphors also limits awareness of how code operates in and for itself and what may be achieved through that. Arguably it is the changes in low-level systems that have provoked the biggest paradigm shifts, such as the development of binary computation and Turing machines [...]" (Yuill, 2004)

To me, this also means that an active engagement with different levels of programming is necessary to

reflect essential aspects of computation. A critical practice around software should therefore not only make use of one specific programming language. This helps to free oneself from the dependencies stated above, and makes it possible to engage on different layers, not only the surface.

I continue with the text, and while Kittler describes buying a commercial version of WordPerfect, I remember my old copy of Word that is still installed on my old partition. I go through the folders of the application folder of my second partition and scan through all the apps that I probably haven't used for months. I follow the alphabetical order of the list view and after some programs starting with "N", a folder called Microsoft appears. I double click on the icon of an orange folder and end up in a grid view, containing 6 files and some folders. In-between them: word.exe. The executable file to open Word. I can't execute it on Linux.

→ I AM A CONSUMER, NOT A USER

Nowadays the software that is required to use a machine comes pre-installed and ready to use. Software can be downloaded from centralized marketplaces: app stores. This causes an immense dependence on the producers, who are in return depending on owners of these marketplaces (including their platform framework and policies). These producers have developed an infinite selection of apps, which is another example of the 'explosion' of software that was previously mentioned. The flood of applications causes software to become a mundane occurrence. The danger here is that we take software for granted. When

we have a problem, there is an app for it. Nobody thinks about the possibility of editing software and adjusting it to one's need. This is not only because usually it is not possible to edit the software due to DRM, but also because the average user is not a user anymore. Instead, people are being educated by companies to be consumers instead of users, let alone creators.[3] It is in the companies' interest to make their clients dependent on their product. Therefore companies are not interested in opening up their products, but they are instead locking them up. They are, then, slowly feeding their clients with updates and new fancy features. This is great for users who just need to get their job done and who want to be in contact with technical struggles as little as possible. On the other hand, it means that for one, the use of software is dictated by companies and, secondly, that if you want to engage with your software more in-depth you cannot do so. Often, you cannot look at the source code, reuse parts of it or modify the program to your needs.

Of course, there is also software that embraces an open and reusable character. This also provides an excellent source for discussion about software. The problem is that such software often requires other programs and more technical knowledge. The average user is not willing to invest that kind of effort. There are also other kinds of software, that

3 The definition of user has changed through the years. In the beginning of computation there was basically no distinction between a user and a programmer, because of the simple fact that users had to program.

embrace the user as an active agent, while still enabling a simple use on the surface. For example, the MediaWiki software allows for accessible editing on the browser, while still providing an infrastructure to extend the functions easily.

"The accompanying paperware" (Kittler, 1992, p. 148) – which paperware? Where is the manual of my document viewer? I move my mouse towards the options on top of the window and click on help. A small window opens, displaying a table of contents. "How to use it", "Find text in documents"… A page containing hyperlinks for different sections. It is probably the first time I ever entered this space of the program.

→ **TELL ME WHAT TO DO**

Software can be so abstract, that the way software affects people is often through the metaphors it uses. What we remember is the animal icon on the start screen, not the algorithm that it uses. For an artistic engagement, I think it is crucial, to carefully examine the different parts of software and then reflect on their use – like, for example, the user manual.

The manual of most programs is part of the software. Actually, the manual is software. The handbook does not come in a physical form anymore, just as the software does not ship on Floppy or CD-ROMs. Software is a download (or a service, that is only running online[4]), so the users cannot touch

4 The concept of software as a service (SaaS) is a very current issue in software. The software is not running on the computer of the user but

it anymore. Thus, it becomes even more abstract. Through the handbook, the software manifests itself as a tool. A tool that has certain functions and the manual describes how to use those functions correctly. Nowadays, the handbook often constitutes a space that stays undiscovered. If we want to consider software as an artistic material, the handbook can further gain new functions as a description, as a space for thoughts. The handbook was also used as a metaphor at the readme festival 2005 to guide visitors through an exhibition of software. Software often remains invisible in its functions and statements, so it is necessary to describe what it is doing. The manual illustrates the fact that the 'user' needs to be informed about what to do with software and how to use it.

I close the help window and find my way back to the text. In the meantime, Kittler turns towards his punchline: there is no software.

→ THERE IS SOFTWARE, CAN'T YOU SEE?
Although software is dependent on hardware, it does not mean that there is no software. A deeper engagement with software means taking software seriously. Even though it might be argued that software is only the representation of machine

rather on the server of the provider. This means that the user does need an internet connection and is constantly sending data to the server. In addition, the user is not in hold of any executable file or program anymore, ending up in even more issues around dependencies.

operations, it is vital to acknowledge software as an independent object of study.

Even though Kittler was arguing that there is no software because it is intrinsically connected to its hardware, Cramer points out that "if any algorithm can be executed mentally, as it was common before computers were invented, then of course software can exist and run without hardware" (Cramer, 2002a).

Following this argument, it points to the idea of software in a very conceptual way, not only defining software as a program that is running on particular hardware. A recent example of this would be that people compute blockchains by hand to demonstrate the math involved. Eventually, all layers of diminishing abstraction on top of hardware deserve attention. Still, it is important to recognize both of the perspectives for their importance – the materialistic and the cultural/political.

There is no clear border between software and hardware. Where does software begin and hardware end? Is it when the code is being compiled, or is it when the machine code is transformed into electrical signals? In the end, the exact point where the software transforms into hardware is not clearly perceivable (Tenen, 2017).

There is undoubtedly a tension between the development of software and hardware. The hardware limits the software. We can not build applications

that run faster than the hardware. Machine learning algorithms, for example, need a lot of resources to calculate their models. This means that effective research with this technology is only possible with sufficient hardware. Even though software can be seen as a conceptual good, it is impossible to execute it only mentally, especially when using very complicated algorithms. Software is only effective through its execution, and thus through its performance.

I continue with reading Kittler. "First, on an intentionally superficial level, perfect graphic user interfaces, since they dispense with writing itself, hide a whole machine from its users" (Kittler, 1992, p. 149).

→ USER INTERFACE

The user interface enables a convenient way to display software (or at least parts of it). This representation is, however, only an interpretation of what the designer thought is the best way to display it (Hadler et al., 2016). At the same time, it looks like this user interface is the only truth that the program holds. It certainly does not become evident that this interface is not neutral. The GUI instead hides. It hides the processes, many functions, the source code, the possibilities, or the decision it takes for you.

The need for a human approach to software also becomes visible from the great use of Graphical User Interfaces. The so-called GUI is not part of the original imaginary of computation, where

> commands were being filled in via a command line. However, today's average user is only surrounded by software displayed via a 'window', encountering the terminal only by chance. Not only does the GUI simplify commands into buttons and mouse-actions, but it also makes software more human. A button that has a 3D effect, the on/off function is displayed via a switch, the mouse transforms into a hand or the form that looks like a letter, which, of course, you fill in by pressing a pen symbol (Fuller, 2008). This is also known as skeuomorphism. It means that objects of the real world are being used to represent digital functions or interface objects. Humans anthropomorphize and use metaphors to communicate the complexities of a less well-known domain (the digital) via the vocabulary and concepts associated with a well-known domain (the physical world). The skeuomorphism in GUIs is an excellent example of that.

As I go further in Kittler's text, focusing on the text, my mail software wants to interrupt me with some notifications about incoming mails. I click them away. Kittler is writing about how computers are writing and reading themselves. I want to copy this part in my notes. I drag the mouse from "in contrast" to "read and write by themselves" and, as the text tints, the layer of text reads: "in contnast to all histor- ical writigtools, are able to read and write by thenvselves" (sic! by ocr?) (Kittler, 1992, p. 147). My machine has read the text before me – not only once. The text has probably been written and read many times before I opened it. The computer had read the document looking for words using Optical Character Recognition, and even made its own interpretation. That explains why the

selected text is wrong because the program misinterpreted some of the characters. Together with this incorrect version of the text, it got written again to the memory. Then, the text was read another time – into the working memory, when I opened it with the document viewer.

→ ERROR: HEADLINE NOT FOUND

We can get a spark of what execution of code means and how software really acts and performs when it fails or when it is taken out of its context (Winograd and Flores, 1995). In the following section, I want to argue that for a serious engagement with software it is also necessary to look at the non-functional and the stuff that is in-between the pixels and conducting paths. We expect software to run seamlessly, but what if software fails or malfunctions? What if software has no function?

"Most people notice infrastructures only when they are put in the wrong place or break down. This means that public knowledge of them is largely limited to their misplacement or malfunction." (Parks, 2009)

While The Alliance for Code Excellence imagines "[a] world where software runs cleanly and correctly as it simplifies, enhances and enriches our everyday life is achievable" (Constant, 2018, p. 11). I argue that the malfunctioning of code can also be something positive that is revealing and holds a value. The interruption of a seamless flow makes undeniable apparent, what could not be seen before. We can use things without being immediately

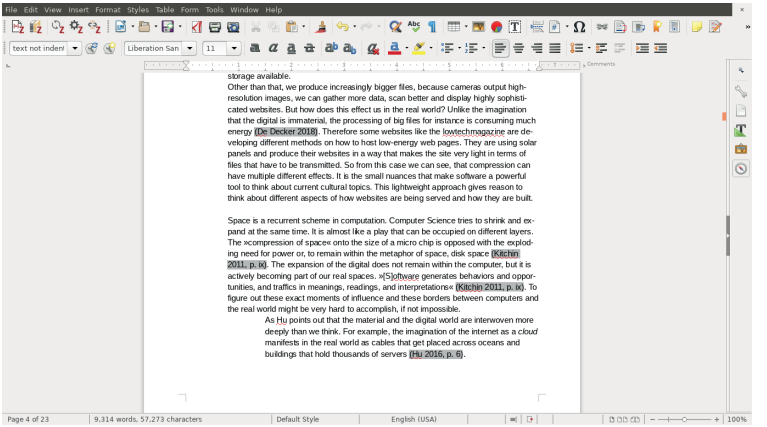

Fig. 4: The writing of this essay with LibreOffice.

aware of their presence, but the 'breakdown' makes them visible. So the malfunctions "reveal[e] to us the nature of our practices and equipment, making them 'present-to-hand' to us, perhaps for the first time" (Winograd and Flores, 1995, pp. 77–78).

For example, the wrong character recognition as visible from the text above can also show how the algorithm works. The mistaken 'm' for 'rn' shows that the algorithm might work with visual comparison and has probably not recognized the gap between 'r' and 'n' – due to the grain of the text. This consequently gives a clue that the algorithm doesn't have an idea about the context of words. Otherwise, it would have figured out that some words are not correct English words.

The way software is set up can embrace the fact that software is failing or not. In the case of seamless software that tries to hide failure, the user does not get any insight. In contrast, when the setup is embracing its unstable character, the user knows that there is a potential for crashes. It means that engagement is undeniable. At the point when it crashes, you are able to get a glimpse of the inner workings of software and possibly be even able to fix it.

The imperfection of software: Digital systems are often considered to be perfect, without the inconsistency and noise that analog systems contain. Instead, digital applications also become inaccurate.

> This is also a result of the dependencies and glitches as pointed out before. Software can even have the same noise as non-digital objects have. When Casey Reas wrote about the new Processing[5], he pointed out the high precision that computers have compared to similar art-forms like Sol LeWitt practiced[6].
>
> "Machines can draw lines with absolute precision so all the imperfections in a physical drawing are removed, giving the rendering different characteristics than those intended by LeWitt" (Reas, 2019). In reality, it turned out that after a few month processing produced the same inaccuracies (glitches) as a drawing by LeWitt would show. This was due to updates and changes in the language.

I change from the document view into the writing program LibreOffice, where I store most of my notes. With a single click on the icon, no keystroke required, the execution starts and the start screen appears. Many processes get triggered by this simple action and the computer follows its instructions, which I do not know – and don't even see. But not with ease this time. The only thing that I can occupy right now, that the process must have 'frozen'. As my mouse indicates with

---

5 A software framework to make programming more accessible for artists.

6 Sol LeWitt was famously known for giving painters instructions on how to draw paintings. His work is also often used as a reference for digital art that follows formal instructions, just like software does, for instance.

a spinning motion, I am unable to continue. I am unable to change the program, I am stuck, just like my program. I try clicking on the icon, again and again, as if my actions would trigger the program to finally make it. It is as if I want to tell the program to try harder by clicking harder. Once again I try to encourage the app, by clicking somewhere randomly on the screen. I give up. I have had this before, so I know how to act. 'kill'[W]. I change to the terminal, type 'sudo killall libreoffice'. I give my permission by typing in my password and, happily, I can see the terminal taking action. With a flicker, the startup screen that was stuck disappears, freeing me and my cursor from redundant spinning. I try restarting the program and hope, that the crash was only due to unlucky circumstances, maybe just something 'got stuck'.

→ IMAGINATIVE SOFTWARE

The perception of software is anything but neutral. Software tells stories, through its metaphors, its contents, its performance. The digital medium offers new ways of telling stories. This becomes obvious not only due to different structures, like the form of the database as Lev Manovich points out but also because of the different modes of intervention software takes in our life (Manovich, 1999). The medium keeps evolving at inexorable speed and so does software, leaving space for new ways of how to tell and what to tell about computation.

That humans tend to anthropomorphize not only their surroundings, but also computers and technology, in general, has been a well-researched topic among computer sciences & psychology. Among others, *The media equation* had shown,

that we as humans consciously and unconsciously anthropomorphize computers (Reeves and Nass, 2003). In addition to that, humans have a vivid and diverse imagination about invisible processes. This includes software. Often, digital media black-boxes certain processes and therefore provides much space for imagination and narratives that can be constructed around it (Finn, 2017). Narratives have been used for the purpose of marketing, and there have been attempts to create relatable stories within applications. A popular example is Joseph Weizenbaum's *Eliza*, a digital application which acted as a therapist, chatting with the user. This piece of software gave convincing proof of how humans anthropomorphize even simple digital applications (Wardrip-Fruin, 2012). Tech giants have put great effort into implementing relatable characters into their systems, e. g. voice assistants. An assistant that is helpful and funny that gathers your data with great pleasure. However, in the past there have also been unsuccessful attempts to add anthropomorphizing elements to programs, only to remind us quickly about Microsoft's famous Clippy (Cain, 2017). These stories in applications and around them make technology more understandable, but can also be a source for misconceptions. A current example seems to be the fear of singularity after machine learning enables applications to 'magically' generate or label images. The gap between the real potential and the imagination around it is significant. I don't want to support an uncritical or blind approach towards technology – I think it is important to be realistic, critical and playful equally

Fig. 5: Software error at McDonalds Regensburg, Germany.

with these algorithms, only then turns engagement into insight.

Another case of narratives is the narrative that exists outside the software. It lies in its performance. How it acts, where and when. The realization that people relate to software on an emotional level makes it possible to create software that tells more than its function. It's possible to tell stories only by how software works. This kind of narrative has been used in some works of Software Art. For example, a work by Luca Bertini which can be found on runme.org. The work is about two viruses in love. "They search for each other on the net, running through connected computers" (Bertini, 2019).

I restart LibreOffice – this time it works. An empty document opens, and a blinking cursor indicates that I am ready to type. I switch back to the text viewer where Kittler's text is waiting for me, and I copy the last sentence. After clicking my way back into my editor, I paste the string from the clipboard to my empty document. Immediately, the text fills the screen:

".Theinversestrategyofmaximizingnoisewouldnot only find the way back from IBM to Shannon, it may well be the only way to enter that body of real numbers originallyknownaschaos". My computer completely rewrote the original text.

→ NOISE IN SOFTWARE

The polished interface makes us forget about what programmers struggle with every day: the noise that surrounds computation. It is the same noise that should make us aware of how imperfect

> and subjective software is, but in many cases, this noise is being suppressed. Every small glitch is being removed out of software and every irregularity is considered a bug. All this noise might instead be the possibility to explore new opportunities with code and its execution further. Maybe the beauty of software lies in exactly this noise, that is being forgotten about in between the logical operations with 0s and 1s.

I save the file and the machine once again writes for me to the hard-drive. I store it using the file format xml. The file gets stored using the name *NotesOnKittler.xml* into the Documents folder. If I open the text in a normal text editor, it turns out. the computer has written noise around the actual text that I saved. This noise makes up the standards of the .xml format, encoding information within <tags>.

2. I AM WRITING, THE COMPUTER IS WRITING

or how to create software

I close all the open windows by pressing CTRL and "W" repeatedly. I open LibreOffice and start writing – the computer starts writing for me. I am typing this very text and the computer constantly listens, displays and saves.

> Note: The arrow → marks a relation to one of the texts above
>
> In the chapter 'The Culture Industry' in *Dialectic of Enlightenment,* Adorno and Horkheimer state that culture is infecting everything with sameness (Adorno, Horkheimer, 2007). They point out how culture has become part of mass production and standardization. Adorno and Horkheimer argue that commercial marketing of culture robs people's imagination and takes over their thinking (idem, p. 98). As an example, they name the transition from telephone to the radio. While everyone was able to communicate through the telephone, the radio transformed the once free actor into a mere listener (idem, p. 115). The reason for this is the commercialization of culture and the resulting relation between consumer and industry. This means that the industry creates culture solely for profits. The consequence for art is that it is also only a product and therefore loses its critical factor and its autonomy (idem, p. 147).

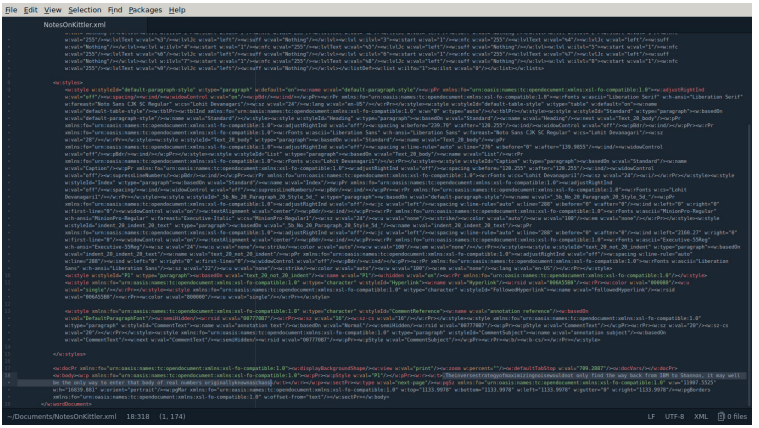

Fig. 6: XML file created with LibreOffice opened with the text editor Atom.

While I disagree with their suggestion to divide art strictly into commercial and authentic art, I think their examination of commercialized culture provides a great observation that also holds true for software in its current state. This is especially true if we consider software as a cultural object, which has extensively been demonstrated by research fields like for example Software Studies. In the following section, I want to argue that (1) the commercial marketing of software contributes to the problems pointed out in the first part of this essay and that (2) artistic methods, as used in Software Art, can provide a potential to counteract and provide a new perspective to these issues.

(1) CULTURE INDUSTRY AND SOFTWARE

Obviously, most software exists to provide financial benefits to the creators (and owners of marketplaces). Also, software has become an object of the culture industry. This means that the user is a consumer (→ I am a consumer, not a user), and that software is guiding people's thinking thus limiting their imagination. Software is made to be used by as many people as possible. Therefore, it has to be simple and generic. Additionally, people are made dependent on proprietary software to get as much money out of them as possible.

Art is confronted with this issue on two different layers. Artist that use software not only have to deal with the limited, commercialized software, but are also taking part in cultural production activly by distributing their art. Following Adorno

and Horkheimer's assumption that art has lost its critical character through commercialization this means that, firstly, the software limits the art or the practice of the artist, and secondly, that the output of the artist is again constrained, through their commercialised artwork. So, instead of criticizing the condition they work in, artists potentially amplify the effect of mass produced culture through the use of commercialized software.

These two layers can be seen from the superficial use of software. The User Interface (→ User Interface) dominates the perception of software for most users. A lot of digital arts, like Generative Art, are focused on output and mostly consider software as a tool (Galanter, 2003). They take over the focus on the surface into their practice. The artistic use of machine learning is a great example of a user that is 'stuck' on the interface layer. Instead of engaging with the inner functions of neural networks, artists generate obscure images, while mostly talking about rather popular topics like datasets, utopia or dystopia (Greene, 2018). The deep dream[7] is not deep indeed. The use of these algorithms is very flat and mostly concentrates only on the output (which is easy to sell). These morphed images are being generated on high-resources machines using libraries that are provided by for-profit corporations (→ Framework culture). This creates a dependence on fast computers and on libraries of third parties. Furthermore, when using those libraries, the user

7 Deep Dream is the name of a machine learning method.

is obedient to the big companies creating such frameworks. Additionally, it also obscures the processes and hides software once more, affirming its already hidden character. Thus, Generative Art doesn't really find a way out of the limitations that come with commercial software, being caught in exactly that loop of cultural production that Adorno and Horkheimer criticize.

(2) SOFTWARE ART AND ARTISTIC METHODS IN SOFTWARE

In the following section, I want to suggest artistic methods that carry the potential to counteract the issues discussed in the previous paragraph. The approach of Software Art provides a great example. Software Art describes the "artistic preoccupation with software production" (Cox, 2007, p. 147). This means that Software Art is using either the software itself or code as its material. The subjects it addresses are mostly the cultural concepts of software (Cramer, 2002b). Software Art does not take software for granted and, therefore, it also acknowledges the importance of the creation process of software (ibid.).

To put focus on the process instead of the end product is not new in the art world, but Software Art exemplifies this approach "appropriate to contemporary conditions" (Cox, 2007, p. 147). This creates the possibility to think of software in terms of performance. While the result is not necessarily a fixed product that is visible, it can be a runtime application, which never reaches the state

of finishing. An approach like this opens up new discussions and new ideas. An example of this is the application *Every Icon* by John F Simon Jr. It is a simple 32 by 32 grid that iterates through every possible combination of black and white squares in the grid. The application has been running since January 14, 1997 and will continue for many years. The application only becomes visible when you visit the website, which displays the current state. Other than that, it performs on its own, reaching formations that will never be seen. In an elegant way this work challenges the viewer's imagination (→ Imaginative Software) about limitations of computation, while automatically producing new, unique images.

Only if the role of software itself is questioned or at least acknowledged in the creation of artworks can the creativity be freed again from the dependencies (→ I depend on you, who do you depend on?) of the culture industry, as pointed out above. This method of Software Art is helpful for both the artist and the user that receives the artwork. Firstly, it opens new ways for the artist to work with software and secondly, the recipient will gain a different perspective on software.

We can see from this that art is occupying practices that can be useful to evoke critical insight into software. Art has shown in its history that it can research complex and abstract issues and deliver critical insights for its recipients. Software is so complex in its relations and so versatile in its

effects that it might be hard to go about a structured analysis. Instead, art might provide a field of exploration and experimentation, which can question and enrich the culture around software. Artistic practice can occupy fields that are difficult to understand on a solely rational level, like in the field of literacy or theater. Brenda Laurel describes computers as a theatre, due to the factors of runtime, interaction and space (Laurel, 2013). The execution of software can be seen as a performance. When the program is executed, machine code turns into machine actions. Software can create emotions and art is able to elicit them. Art offers the opportunity to deeply engage with certain aspects of software and connect the cultural to the scientific realm. Software creates new ways of expression for artists. Artists can generate experimental software, that doesn't need to function as a program but can work as a cultural object, a critique or a question. Art has famously shown that it can re-contextualize and question objects of our daily life, that have become invisible. Art can make software visible again (→ There is software, can't you see?), and question how software is used. It is therefore crucially important for artists who work with software to understand its internal functioning.

The focus on software was the strength of Software Art, which unfortunately got lost in the past years. Software Artists dissolved to other fields which are often less specific, like new media arts or algorithmic art. Ultimately, I want to encourage a rediscovery of the methods of Software Art and a

new engagement with software in the arts, which has the potential to educate users to be more critical about their software usage.

CONCLUSION

The artistic methods and impulses provided above should encourage a way of thinking that reflects on the inner functions of software, aiming to remap the issues that we currently see in the realm of cultural production of software. Deeper artistic engagement with software is promising to find a balance between beautiful artistic expression and fundamental discussion around software usage and production. Poetic Software can be playful and serious, subjective and emotional, inspiring and revealing, helpful and funny at the same time. What is important is that it shows genuine engagement, while not falling into the trap of commercial and thoughtless software usage.

It might be a way to emphasizes the glitch (→ ERROR: headline not found) or use the narrative (→ Imaginative Software), to show what software is not, and what else software could be.

Eventually, this might be the way back to the noise (→ Noise in Software) that Kittler was calling for.

I click save and close LibreOffice. The operating system kills the process and shows the empty desktop.

REFERENCES

Anon. (2018) The Open Group Base Specifications Issue 7, 2018 edition, [online]. Available at: http://pubs.opengroup.org/onlinepubs/9699919799/utilities/kill.html

Bertini, L. (2019) *runme.org – say it with software art!* [online]. Available at: http://runme.org/project/+ViCon/

Cain, A. (2017) *The Life and Death of Microsoft Clippy, the Paper Clip the World Loved to Hate* [online]. Artsy. Available at: https://www.artsy.net/article/artsy-editorial-life-death-microsoft-clippy-paper-clip-loved-hate

Constant. (2018) *The Techno-Galactic Guide to Software Observation*.

Cox, G. (2007) Generator: The Value of Software Art. *In*: Rugg, J. and Sedgwick, M., eds. *Issues in Curating Contemporary Art and Performance*. Intellect Books, 147–162.

Cramer, F. (2002) *Concepts, Notations, Software, Art* [online]. Available at: http://cramer.pleintekst.nl/essays/concept_notations_software_art/concepts_notations_software_art.html

Cramer, F. (2002b) Contextualizing Software Art, 9.

Davis, F. (2016) We shape our tools and, thereafter, our tools shape us. *Medium* [online]. Available at: https://medium.com/@freddavis/we-shape-our-tools-and-thereafter-our-tools-shape-us-1a564cb87484

De Decker, K. (2018) *How to Build a Low-tech Website?* [online]. LOW←TECH MAGAZINE. Available at: https://solar.lowtechmagazine.com/2018/09/how-to-build-a-lowtech-website.html

Finn, E. (2017) *What Algorithms Want: Imagination in the Age of Computing*. Cambridge, MA: The MIT Press.

Fuller, M. (2008) *Software Studies – A Lexicon*. Cambridge, Mass: The MIT Press.

Galanter, P. (2003) *What is Generative Art? Complexity Theory as a Context for Art Theory* [online]. Available t: https://www.philipgalanter.com/downloads/ga2003_paper.pdf

Google, M. (2019) *Calibration Target* [online]. Google. Available at: https://www.google.com/maps/@34.8511698,-117.6572542,103m/data=!3m1!1e3

Greene, T. (2018) *Someone paid $432K for art generated by an open-source neural network* [online]. The Next Web. Available at: https://thenextweb.com/artificial-intelligence/2018/10/25/someone-paid-432k-for-art-generated-by-an-open-source-neural-network/

Hadler, F., Haupt, J., Andrews, T. L., Callander, A., Flender, K. W., Haensch, K. D., Hartmann, L. F., Hegel, F., Irrgang, D., Jahn, C., Lialina, O., Szydlowski, K., Wirth, S., and Yoran, G. F. (2016) *Interface Critique*. Berlin: Kulturverlag Kadmos Berlin.

Horkheimer, M. and Adorno, T. W. (2007) *Dialectic of Enlightenment*. 1 edition. Stanford, Calif: Stanford University Press.

Hu, T.-H. (2016) *A Prehistory of the Cloud*. Reprint edition. Cambridge, Massachusetts: MIT Press.

Kitchin, R. (2011) *Code/Space – Software and Everyday Life*. Cambridge, Mass: MIT Press.

Kittler, F. (1992) There is no Software. [online]. Available at: https://monoskop.org/images/f/f9/Kittler_Friedrich_1992_1997_There_Is_No_Software.pdf

Laurel, B. (2013) *Computers as Theatre*. 2nd ed. Upper Saddle River, NJ: Addison-Wesley Professional.

Manovich, L. (1999) Database as Symbolic Form. *Convergence*, 5 (2), 80–99.

Mansoux, A. (2017) Sandbox Culture: A Study of the Application of Free and Open Source Software Licensing Ideas to Art and Cultural Production. [online]. Available at: https://monoskop.org/images/e/ea/Mansoux_Aymeric_Sandbox_Culture_A_Study_of_the_Application_of_FLOSS_Licensing_Ideas_to_Art_and_Cultural_Production_2017.pdf

Parks, L. (2009) Around the Antenna Tree: The Politics of Infrastructural Visibility. [online]. Available at: http://www.flowjournal.org/2009/03/around-the-antenna-tree-the-politics-of-infrastructural-visibilitylisa-parks-uc-santa-barbara/

Reas, C. (2019) *{Software} Structures by Casey Reas et al.* [online]. Available at: https://artport.whitney.org/commissions/softwarestructures/text.html

Reeves, B. and Nass, C. (2003) *The Media Equation: How People Treat Computers, Television, and New Media Like Real People and Places*. Reprint. Stanford, Calif: Center for the Study of Language and Inf.

Tenen, D. (2017) *Plain Text: The Poetics of Computation*. Stanford University Press.

Ullman, E. (2012) *Close to the Machine*. Reprint edition. New York: Picador Paper.

Wardrip-Fruin, N. (2012) *Expressive Processing: Digital Fictions, Computer Games, and Software Studies*. Cambridge, Mass London: The MIT Press.

Winograd, T. A. and Flores, F. (1995) *Understanding Computers and Cognition: A New Foundation for Design*. New ed. Boston: Addison Wesley.

WinZip Computing, S.L. (2010) *WinZip Logo* [online]. Available from: https://www.pc-magazin.de/bilder/8610089/800x480-c2/WinZip-Logo-Aufmacher.jpg

Yuill, S. (2004) Code Art Brutalism: Low-Level Systems and Simple Programs. *In*: *read_me, Software Art and Cultures*. Aarhus: Aarhus University Press, 120–163.