Command and Ctrl: How Digital Became Us

GALO CANIZARES
The Ohio State University

Looking at the politics of software, this paper proposes that computation was never free from cultural bias. Early computation embedded military, workplace, and domestic values into its structures, and as the field professionalized itself during the late 20th century, it excluded input from diverse social groups. Software eventually became a mechanism that, under the guise of user-friendliness, obfuscates more than it reveals.

If "becoming digital" suggests that architecture has embraced and is inseparable from computation and software, then the logical next step is to identify the limits and merits of such a symbiosis. While scholarship during the previous decade primarily addressed technical means through which design disciplines became digital, this paper will introduce the political dimension of software. Referencing the work of Wendy Hui Kyong Chun, Lev Manovich, and other media theorists, I will argue that the systems used to produce architectural media are neither benign nor unbiased platforms that are active participants in the design process. Humans did not simply become digital, but the digital also became human. Embedded within these machines are all of our flaws and understandings of the world, be it aesthetic, technical, or political. In other words, software's role in design could be regarded as ideological and therefore warrants a critical approach.

COMMAND AND CTRL

> While we are indeed 'being digital,' the actual forms of this 'being' come from software.
>
> —Lev Manovich[1]
>
> Interfaces determine user behaviors by projecting illusions that are apparently neutral and have evidently good intentions, such as clearing a path through a darkly unknowable space for the movements of the will.
>
> —Ryan Kuo

Is software political? And if so, what parts of it are political? Historically, software has eluded critical scrutiny due to its complexity and opaqueness; computers are typically regarded as black box instruments that reveal only what is helpful for users. Yet, as we are fully aware, software pervades all aspects of daily life from the time our alarms wake us up to the last email we send out. Given this reality, the power dynamics at play between computer programmers, technology corporations, and users must be a part of contemporary discourse on the subject. These power dynamics constitute what I call the politics of software: a flux of sovereign relationships that exists throughout the digital milieu. Its participants range from ideological figures (your Steve Jobses, Bill Gateses, Elon Musks, etc.) to GUI designers to end-users. Most of us will undoubtedly fit in the last category since, as software users, we continuously hand over a lot of power to interfaces. Sometimes this takes the form of lengthy user agreements or privacy statements, but more often than not the exchange of sovereignty in the digital realm takes place at the point of purchase: when we select which tool to use. The decision to choose one software over another is a political decision that inevitably shapes the product.

How then should architects and designers who work primarily with software engage digital tools on a political level, beyond the technological or the representational? For many in design professions and design education, work is synonymous with specific brands of software. As Lev Manovich points out in *Software Takes Command*:

> The new ways of media access, distribution, analysis, generation, and manipulation all come from *software*. Which also means that they are the result of the particular choices made by individuals, companies, and consortiums who develop software…[2]

These choices, I argue, result in discrete allegiances and a kind of blind faith. For example: while most users do not need to know exactly how image editing software manipulates image kernels, they would agree that specific applications have become part of our everyday vocabulary. Adobe Photoshop, for example, "has become parlance for all graphic design software and for the act of manipulating digital images in general."[3] We say images are *Photoshopped* when we really mean *digitally manipulated*. Yet despite the trademark transmogrifying itself into a colloquial verb synonymous with image manipulation, Adobe maintains that using Photoshop as a verb or noun violates its terms of use.[4] This push and pull between those who own and develop software, and those who own and design *with* software has produced a variety of user types: pirates, hackers, open-source die-hards, Mac yuppies, PC gamers, Linux iconoclasts, etc.; in short, a generation of users whose design methods, vocabularies, and

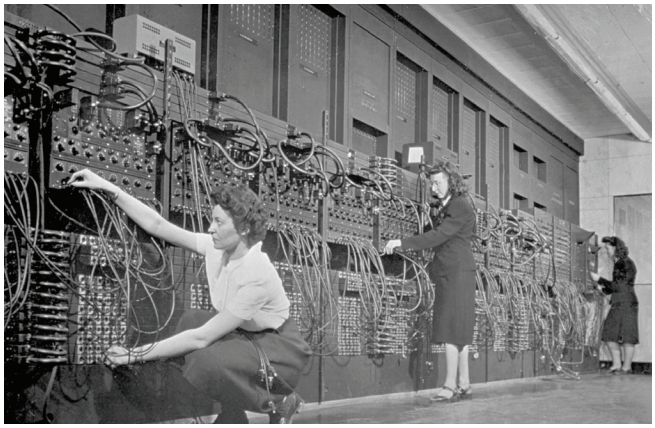

Figure 1. "Computers" working on the ENIAC machine at the University of Pennsylvania, 1945.

rituals are inseparable from the proprietary tools used in the creative process.

Architectural historian Mark Jarzombek describes this phenomenon as a new ontology. In *Digital Stockholm Syndrome*, he argues that questions of being are today inextricably linked to our "data exhaust:" the outputs of our interactions with software. Whether explicitly or not, this "data exhaust... is meticulously scrutinized, packaged, formatted, processed, sold, and resold to come back to us in the form of entertainment, social media, apps, health insurance, clickbait, data contracts, and the like."[5] According to Jarzombek, this not only characterizes who we are as *users*, but also how we conduct ourselves as *beings*; two terms growing increasingly inseparable.

Beings becoming users also poses a political problem that crosses disciplinary boundaries. What the Photoshop-as-verb example illustrates is how software is taken for granted in creative fields. Of course, media programs are often used politically to produce imagery and propaganda, but what of the programs themselves? Can we effectively say that the applications used to depict existing images as well as bring new elements into the world are unbiased themselves? How do we discern between a corporate ideology and the software it pushes? If Macs and PCs can effectively divide up the world into digital camps, how else is software conditioning its users?

Though most rhetoric that criticizes power relationships tends to favor the abolition of such structures, the argument here is instead that close scrutiny of the biases and blind spots of software would: (1) empower users (in our case, architects and designers) to better utilize or select their tools, (2) enable users to contribute to software design, (3) expand the critical discourse of these obscure technologies, and (4) support a skepticism of systems that obfuscate more than they reveal.

In order to begin, however, we must first look at some early episodes of computation, where real-world politics and computation were first imbricated.

EARLY DAYS

Politics[6] and computation have a long history together. Modern computation grew out of World War II, specifically from military decryption projects such as the Bombe and Enigma along with the pioneering mathematics of figures like Alan Turing and John von Neumann. During these crucial years, various power dynamics of military protocol, command and control mechanisms, and other hierarchies would be embedded into the language of computation. A perfect example might be the term computer itself, which used to designate primarily female clerks tasked with numerical calculations. After 1945, the term rarely applied to humans, but maintained its initial set of hierarchies.[7] Computers became man's (male emphasis on purpose) subservient machines, entrusted to universities for further research, while remaining owned and operated by various military sponsors.

Given the complex military and male-dominated frameworks into which these early computers were introduced, it would be difficult to call their structures unbiased. As various media scholars have noted, these systems not only produced innovative engineering solutions, but also codified master-slave relationships and biased operating procedures.[8] For instance, while Howard Aiken, naval commander and inventor of the Harvard Mark I computer, established some of the first protocols for using a computer, his own personal values inevitably became part of those protocols. Aiken infamously ran his laboratory like a naval facility, requiring that all employees "show up in full uniform and call him 'commander'" and would refer to the computer as "a 'she,' like any navy ship."[9] Turing, on the other hand, would refer to computers—both human and machine—as slaves, going so far as to state that: "the intention in constructing these machines in the first instance is to treat them as slaves, giving them only jobs which have been thought out in detail."[10] As a result, both "master" and "slave" are still terms embedded in the language of computation, particularly in device communication hierarchies, and Aiken's feminization of the computer can readily be seen in today's attitudes towards digital devices such as digital assistants and GPS interfaces.[11]

Along with Turing's and Aiken's visions for subservient machines, Wendy Hui Kyong Chun reminds us that, in the 1940s, "programming proper" was defined as "a man sitting at a desk giving commands to a female 'operator,'" suggesting that in spite of a desire to innovate, computation and the design of computing interfaces merely reflected traditional office structures.[12] Though female analysts and programmers existed - most notably Grace Murray Hopper, who led the Mark I (and II) system team with Aiken - the act of

The Computer Girls

BY LOIS MANDEL

A trainee gets $8,000 a year ...a girl "senior systems analyst" gets $20,000—and up! Maybe it's time to investigate....

Ann Richardson, IBM systems engineer, designs a bridge via computer. Above (left) she checks her facts with fellow systems engineer, Marvin V. Fuchs. Right, she feeds facts into the computer. Below, Ann demonstrates on a viewing screen how her facts designed the bridge, and makes changes with a "light pen."

Twenty years ago, a girl could be a secretary, a school teacher . . . maybe a librarian, a social worker or a nurse. If she was really ambitious, she could go into the professions and compete with men . . . usually working harder and longer to earn less pay for the same job.

Now have come the big, dazzling computers—and a whole new kind of work for women: programming. Telling the miracle machines what to do and how to do it. Anything from predicting the weather to sending out billing notices from the local department store.

And if it doesn't sound like woman's work—well, it just is.

("I had this idea I'd be standing at a big machine and pressing buttons all day long," says a girl who programs for a Los Angeles bank. I couldn't have been further off the track. I figure out how the computer can solve a problem, and then instruct the machine to do it."

"It's just like planning a dinner," explains Dr. Grace Hopper, now a staff scientist in systems programming for Univac. (She helped develop the first electronic digital computer, the Eniac, in 1946.) "You have to plan ahead and schedule everything so it's ready when you need it. Programming requires patience and the ability to handle detail. Women are 'naturals' at computer programming."

What she's talking about is *aptitude*—the one most important quality a girl needs to become a programmer. She also needs a keen, logical mind. And if that zeroes out the old Billie Burke-Gracie Allen image of femininity, it's about time, because this is the age of the Computer Girls. There are twenty thousand of them in the United *(cont. on page 54)*

Photos by Henry Grossman. Dress by Gino Charles.

52

Figure 2. Article in ***Cosmopolitan*** magazine, April 1967.

computing remained dependent on "yes, sir" responses to declarative sentences and commands. Through the 1960s, these regimented patterns were also orchestrated spatially. Paul Ceruzzi notes, a typical transaction began by submitting a deck of cards to an operator through a window (to preserve the climate control of the computer room). Sometime later the user went to a place where printer output was delivered and retrieved the chunk of fan-fold paper that contained the results of his or her job.[13]

The interface here was a physical room populated by operators working to enact a series of instructions. This command-and-response structure is not only the root of modern programming languages and their parataxic form, but also the affirmative, passive structure of interfaces in general. It is the reason we do not ask a computer if it can print, but rather give it a print *command*. Eventually, "command and control" became less of an abstract idea and was literalized as a codified action when keyboards introduced the Command (Apple III, 1980) and Ctrl (Teletype Model 33, 1963) keys on computer keyboards. In contrast to other keys, which would output ASCII characters, these were designed to communicate directly with the interface, allowing for quicker initiation of procedures—what we today call "shortcuts."

What these early episodes in computation illustrate is how easily existing power structures can slip into new technologies. Software and hardware are not explicitly designed as biased structures, but through the implementation and orchestration of certain protocols, these neutral machines can reflect the wills of their makers, as well as concurrent political contexts. As a contemporary example, we need only look at machine-learning software, an advanced tool for teaching computers to recognize patterns, faces, or objects in order to execute interpretive actions. In 2016, a University of Virginia machine-learning experiment exposed a behavior pattern in which image sorting software would automatically associate certain images with specific genders (e.g. kitchens with women).[14] Researchers found that the image sets used to train these machines were gender biased themselves, and thus the software reflected that model. Several similar instances have occurred since, involving not just gender but also racial bias.

As advanced as these machines might be, they inevitably reflect, and may even amplify, whatever prevailing cultural values are available to their programmers. The University of Virginia researchers were eventually forced to conclude that software may "not only reinforce existing social biases but actually make them worse."[15] Media scholar Tara McPherson offers a corollary reminder that "digital media were born as much of the Civil Rights era as of the Cold War era."[16] Therefore, whether explicitly or not, these systems have embedded within them layered cultural meanings that must continuously be revisited. Narrowing in on race more specifically, McPherson continues, "we must understand and theorize the deep imbrications of race and digital technology even when our objects of analysis seem not to 'be about' race at all."[17] This is indeed software's political paradox: that as it grows seemingly democratic, it also becomes even more opaque.

LATTER DAYS

> Software and ideology fit each other perfectly because both try to map the material effects of the immaterial and to posit the immaterial through visible cues.
>
> —Wendy Hui Kyong Chun

If the earliest computer systems reflect the wills and methods of the military-industrial complex, then today's systems mirror those of Silicon Valley. One of the earliest ventures in Silicon Valley was the Stanford Research Institute (now SRI), which stood in direct contrast to East Coast research laboratories. SRI sought to free computer scientists from the shackles of government contracts and provide a culture of unfettered creativity. Nowhere was this notion more firmly established than at Xerox PARC (Palo Alto Research Center) where a group of privileged individuals would design the future of computing. PARC was, to put it bluntly, "a place for straight cis white men in business ties to sit on bean bag

Figure 3. A brainstorming session at Xerox PARC.

chairs and embrace consequential ideas without fear of retribution," an image best exemplified by a popular PARC photograph featuring staff from the Computer Science Lab discussing work in a conference room full of bean bags.[18]

Silicon Valley was notably the birthplace of the computer mouse and the GUI, but it was also home to the earliest computer ideologues. In their essay, "Black Gooey Universe," American Artist points out that PARC was the place where the computer screen turned from black to white, an ideological maneuver. They posit, "[t]he transition of the computer interface from a black screen, to the white screen of the 70s, is an apt metaphor for the theft and erasure of blackness, as well as a literal instance of a white ideological mechanism."[19] By framing the history of Silicon Valley's lack of diversity against its technological advancements, Artist produces a reading of software that reveals its biases. Though the shift from command line terminals to GUI—from black screen to white screen—reduced the amount of knowledge required to operate a computer, it also obfuscated its underlying mechanisms. And while the transition was done in the name of "user-friendliness," the price paid was transparency.

As a result of this shift to GUIs, computers now hide most of their protocols and source code in the name of user-friendliness. However, because this approach "obfuscates more than it reveals," it could be said to enable a literal blind faith in operating systems (OS).[20] According to Alexander Galloway, "[s]oftware operates through a technological model that places a great premium on meticulous symbolic declarations and descriptions, yet at the same time requires concealment."[21] Today, software mitigates this concealment by allowing a false sense of ownership; it refers to *your* documents and is tailored to *your* preferences. But underneath this layer, proprietary systems retain ownership, for example, by installing automatic updates or by allowing only specific file extensions. By giving users a certain false agency over their interfaces, software conditions and produces personality types, from "open-source power geeks," to those who "think different." This conditioning is perhaps easiest to see in the OS wars between Mac and PC or iOS and Android. While these wars are less politically than economically charged, they nevertheless symbolize the various lenses through which we view the digital world.

In a way, user-friendliness and obfuscation have become the overarching ideologies of Silicon Valley. Ease of use is often a prominent selling point for software. Even contemporary programming languages are dumbed down, optimized, and packaged so that one does not need to program from scratch, but instead simply assembles pre-built modules. This is evident in programming suites like Python, Grasshopper, and

JavaScript. However, just as we had to be skeptical of Turing's use of master-slave dynamics, we should also be skeptical of software that, as American Artist noted, exists as an echo chamber for white values and beliefs. Modularity, for instance, was an innovative concept in the late 1960s for UNIX operating system designers as well as architects as well as governments responding to social unrest;[22] PARC did not hire a female artist-in-residence until 1993; and user-friendliness was not born out of an ethical consensus between various social groups, but rather from a top-down, white-male, economic drive to sell the most computers. Steve Jobs makes this latter point clear in a 1985 interview:

> People really don't have to understand how computers work. Most people have no concept of how an automatic transmission works, yet they know how to drive a car. You don't have to study physics to understand the laws of motion to drive a car. You don't have to understand any of this stuff to use Macintosh.[23]

With this statement, Jobs was adamantly pushing the Macintosh as both a product and ideology. For Jobs, users were consumers, and the consumer did not require any advanced knowledge; user-friendliness was a commodity.[24]

In a sense, advertising is the clearest litmus test for software's ideological promises. Take, for example, the catchphrase, "There's an App for That," coined by Apple in a 2009 ad campaign. The phrase implores users to download, and developers to design, small applications that may optimize daily life. Since the popularization of the tagline, apps have surpassed boxed software as the primary method for purchasing and installing computer programs. Furthermore, with over 2 million apps on the Apple Store (3.8 million in Google Play), app culture clearly embodies a kind of techno-positivist agenda contingent on the belief that apps may be able to solve any given problem.

FAITH

But faith in computation is not a new phenomenon in architectural design. Around 1962, architect Christopher Alexander used computation to develop a theory of urban design that would oppose both post-war suburbanization and modernist masterplanning. Critical to his research was an IBM 7090 computer, which was used to compute what Alexander would eventually refer to as "design patterns," logical rules for designing spaces from the scale of the room to the city. Perhaps more interesting than the computer he used, was the computer that Alexander did not use, the Lincoln TX-2. In "Alexander's Choice: How Architecture Avoided Computer-Aided Design c. 1962," Alise Upitis tells the story of how Alexander chose the IBM 7090 over the TX-2, which had a graphical display. While Ivan Sutherland famously used the TX-2 to develop the first drawing software Sketchpad, the ancestor of today's 3D

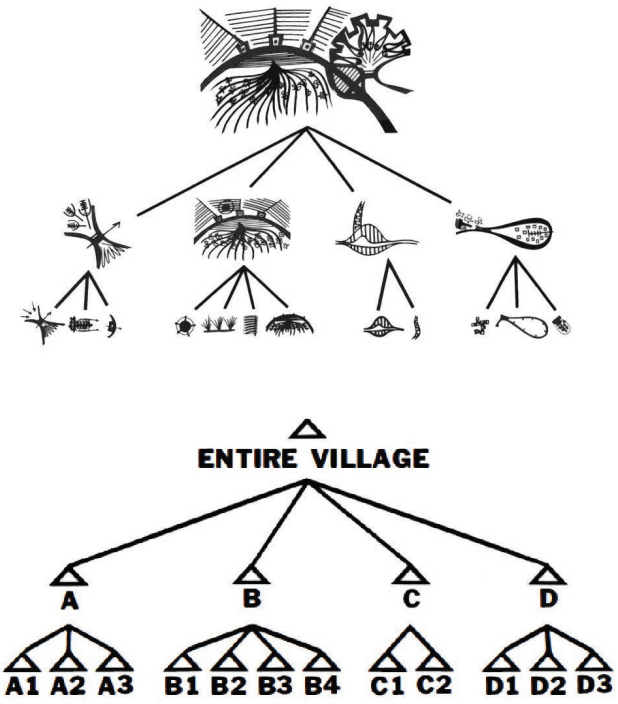

Figure 4. Design patterns from Alexander's book, A Pattern Language.

modeling suites, Alexander focused his efforts on abstract design problems conceived as matrices. His method of reducing design to a set of programmable requirements instead of graphical relationships, Upitis argues, "was determined by the material realities of his chosen technology... the series of IBM mainframes," not by any other preconceived notion.[25] In other words, the computer dictated the method. Alexander's choice not only led to a systems-based approach to city planning, but also paved the way for parametric design and, as Molly Wright Steenson has noted, directly contributed to the development of object-oriented programming languages.[26]

Alexander's case is curious because of the unexpected computational role-reversal. His thought was clearly shaped by the format of the system on which he worked. Steenson again: "Alexander developed visual structures that corresponded to what his computer programs could calculate: trees, semilattices, and networks."[27] His diagrammatic approach to city planning and design problems in general was born out of the machinic logics of the IBM 7090. Of course, systemic thinking was not solely born out of computation; Cybernetics, gestalt psychology, and heuristics (prevalent in the 1960s) had their part to play in Alexander's methodologies. But we cannot ignore the direct correlation between the punch card programs, visual diagrams, and design solutions. While some might say he was thinking computationally, for our purposes we can state that the interface certainly had power over the final output.

Computation affects our thinking simply by dictating the terms of work through direct manipulation of information-laden objects. In the case of design disciplines, these interfaces introduce vocabularies and methods, which were rarely used (or did not even exist) before software. Like the Photoshop-as-verb observation, words such as "pixelated," "hi-res," "raytraced," and "parametric" are perfectly acceptable descriptors of contemporary visual media. As software advances, new visual judgements are continuously extracted from the interface during the design process. In contemporary architectural discussions, buildings are sometimes equated with specific software aesthetics, such as Autodesk Revit; 3d modeling functions are used to describe formal appearances ("lofted" and "booleaned"); and, most strikingly, the noun "drawing" now refers primarily to a digital file than a physical artifact. Software's limitations also shape the appearance of our products; more computational power often equals more geometric complexity and vice versa. Ultimately, like Alexander's pattern languages, the actions enabled by software are literalized by the discipline.

Beginning in 2005, the architecture firm MOS introduced a series of software tools related to their small-scale installations. The accepted narrative is that MOS used these experiments to play around with different entropic effects. By using game-engines and simulation technologies as part of their design process, they were able to introduce new behaviors and characteristics to traditional 3D modeling. The results were stacks, piles, and other aggregations of forms sorted and arranged by simulated gravity, density, elasticity, and other real-world physics. In contrast to parametric modeling, which prioritized novel, formal, and complex geometries, MOS used 3D primitives as their main formal language, relying on forces to do the rest. In the end, the installations and software were—and are still—praised as an alternate approach to architectural composition making/finding.

As a result of their obvious antagonism to the smooth geometry, or *parametric*, project MOS's software and installations engendered a kind of deadpan critique of contemporary architectural practice. They advocated for primitives over complex surfaces, screenshots over glossy renderings, and repetition over novelty. Ironically, the practice eventually became the poster child for an anti-iconographic, pro-generic method of design that is still in vogue among young practices.

A more political interpretation might be that MOS developed their own software as a means of critiquing contemporary architecture's blind faith in computer applications. Knowing that software has so engulfed the discipline as to constrain certain modes of practice, MOS set out to design not only a series of small installations, but their own proprietary apps for the dissemination and production of their architectural media. Bespoke programs could eschew big-name-software's cost, fluff, and intimidatingly labyrinthine user interfaces and workflows. No longer would architects be subject to licensing limitations or proprietary file formats. They could simply develop a facade app, or a block aggregator that could iterate through schemes and produce screenshots, which could be used as construction drawings. In short, instead of architectural design depending on software, it would itself become software.

This ultimate eff-you to powerful figures like Autodesk or Adobe allowed the office to break free from the constraints of proprietary file-formats and concentrate exclusively on the delineation of specific spatial, tectonic, and visual effects. By constructing the source code that would generate gravity and mass, for example, MOS not only followed the tradition of artists mixing their own paints a la Yves Klein or Johannes Vermeer, but in fact exposed the very makeup of the forces they sought to exploit. Through a thorough investigation of code and programming, MOS's software unearthed the mechanisms of entropy. Just as Alexander's patterns proposed new architectural design methods, software-simulated entropy suggested a new path for architectural experimentation. A simple act of adding gravity to a stack of blocks, completely changes its posture and character and, therefore, the code facilitating those forces becomes a perfectly viable architectural material.

For MOS, designing their own software broke open the black box of computation. But their software also exploited the potential for the world of the screen to exist as the medium of architecture beyond building, or the potential for software to be used for its own effects instead of its fidelity to the outside world. For instance, by programming their own applications, they introduced new vocabularies to designers, who until then remained fixated on a language pushed onto architecture by software engineers (*loft*, *extrude*, *sweep*). After MOS, buildings could *droop*, *erode*, *drift*, *crumple*.

Though user-friendly software, operating systems, and applications may not initially seem like ideological tools, they behave in similar ways. In the case of MOS, it is difficult to reconcile whether the software is or isn't dominant over the designer. What is clear is the negotiation between the two parties, which over time becomes a symbiotic process: the app produces unexpected results, then the user reflects on those results. At a larger scale, consider Autodesk Inc's 2018 homepage header: "Ready to Make Anything." It implies that Autodesk software is unbounded. The audacity of this statement makes it more political than informative. It asserts the company's stronghold over certain industries and promotes the instant gratification we have come to expect with real-time computing; it is ready (user-friendly, fast) to make (draw, model, 3D print) anything (limitless creativity).

Software's direct influence on design choices runs parallel to Wendy Chun's final point on the politics of software. Her claim is an extension of Slavoj Zizek's argument that "ideology persists in one's actions rather than in one's beliefs."[28] Because the public cannot fully understand nor control computation, she states, software's significance should be measured by the actions it enables. These actions are, of course, facilitated by interface devices in conjunction with software. Chun argues that software's real-time flow has collapsed the distance between command and execution, resulting in both a democratization of power (everyone has the power to execute commands) and a new perception of ideology (computers are fundamentally ideology machines). She states, "in a formal sense computers are understood as comprising software and hardware are ideology machines. They fulfill almost every formal definition of ideology we have, from ideology as false consciousness (as portrayed in *The Matrix*) to Louis Althusser's definition of ideology as 'a "representation" of the imaginary relation of individuals to their real conditions of existence.'"[29] The interfaces that allow certain actions emerge out of specific problem-solving initiatives; and to identify a problem, it is necessary to establish a position, which is and always will be a political act. And if software represents a kind of ideological framework concerning how these commands should be executed and the scope of its cultural effects, then the devices we use to trigger those actions are themselves extensions of our own political will.

ENDNOTES

1 Lev Manovich, *Software Takes Command* (London: Bloomsbury Academic, 2013), 149.

2 Manovich, 148.

3 Aaron D. Knochel, "Seeing Non-Humans: A Social Ontology of the Visual Technology Photoshop," Ph.D. diss., The Ohio State University, 2011.

4 This sentence itself is an example of a misuse of the trademark. See "General Trademark Guidelines," Adobe official website, accessed August 15, 2018. https://www.adobe.com/legal/permissions/trademarks.html

5 Mark Jarzombek, *Digital Stockholm Syndrome in the Post-Ontological Age* (Minneapolis: University of Minnesota Press, 2016).

6 Politics will be defined here as any activities involving power dynamics.

7 Paul Ceruzzi, *A History of Modern Computing* (Cambridge, MA: The MIT Press, 1998), 1.

8 See texts such as Tara McPherson, "US Operating Systems at Mid-Century: The Intertwining of Race and UNIX," in *Race after the Internet* (Abingdon, UK: Routledge, 2013), 27-43. And Wendy Hui Kyong Chun, "On Software, or the Persistence of Visual Knowledge," *Grey Room* 18 (Winter 2004): 26-51.

9 Claire L. Evans, *Broad Band: The Untold Story of the Women Who Made the Internet* (New York: Portfolio-Penguin, 2018), 34.

10 Alan Turing, "Lecture to the London Mathematical Society," lecture, London, February 20, 1947.

11 In computing, "master" and "slave" are typically used to designate which device or process will be in charge of others in a system. It should be noted that there has been campaigns to rename this relationship differently, for instance, as "primary" and "replica."

12 Chun, "On Software, or the Persistence of Visual Knowledge."

13 Ceruzzi, *A History of Modern Computing*, 77.

14 Tom Simonite, "Machines Taught by Photos Learn a Sexist View of Women," *Wired*, August 21, 2017. https://www.wired.com/story/machines-taught-by-photos-learn-a-sexist-view-of-women/.

15 Simonite, "Machines Taught by Photos Learn a Sexist View of Women."

16 McPherson, "US Operating Systems at Mid-Century," 34.

17 McPherson, "US Operating Systems at Mid-Century," 34

18 American Artist, "Black Gooey Universe" in *Unbag* 2, accessed August 10, 2018. https://unbag.net.

19 American Artist, "Black Gooey Universe."

20 Chun, "On Software, or the Persistence of Visual Knowledge."

21 Alexander R. Galloway, "Language Wants To Be Overlooked: On Software and Ideology," *Journal of Visual Culture* 5, no. 3 (2006): 315-331.

22 McPherson, "US Operating Systems at Mid-Century," 30.

23 David Sheff, "Playboy Interview: Steven Jobs," *Playboy*, February 1985.

24 For a comprehensive analysis of user ideologies see: Olia Lialina and Dragan Espenschied, "Do You Believe in Users?" in *Digital Folklore Reader* (Switzerland: Merz & Solitude, 2009).

25 Alise Upitis, "Alexander's Choice: How Architecture Avoided Computer-Aided Design c. 1962," in *A Second Modernism: MIT, Architecture, and the 'Techno-Social' Moment*," (Cambridge, MA: The MIT Press, 2013), 474-506.

26 Molly Wright Steenson, *Architectural Intelligence: How Designers and Architects Created the Digital Landscape* (Cambridge, MA: The MIT Press, 2017).

27 Steenson, 26.

28 Chun, "On Software, or the Persistence of Visual Knowledge," 44.

29 Chun, "On Software, or the Persistence of Visual Knowledge," 43.