On "Sourcery," or Code as Fetish

Wendy Hui Kyong Chun
Brown University

Abstract

This essay offers a sympathetic interrogation of the move within new media studies toward "software studies." Arguing against theoretical conceptions of programming languages as the ultimate performative utterance, it contends that source code is never simply the source of any action; rather, source code is only source code after the fact: its effectiveness depends on a whole imagined network of machines and humans. This does not mean that source code does nothing, but rather that it serves as a kind of fetish, and that the notion of the user as super agent, buttressed by real-time computation, is the obverse, not the opposite of this "sourcery."

Debates over new media often resonate with the story of the six blind men and the elephant. Each man seizes a portion of the animal and offers a different analogy: the elephant is like a wall, a spear, a snake, a tree, a palm, a rope. Refusing to back down from their positions—based as they are on personal experiences—the wise men then engage in an unending dispute with each "in his own opinion / Exceeding stiff and strong / Though each was partly in the right, / And all were in the wrong!" The moral, according to John Godfrey Saxe's version of this tale, is: "So oft in theologic wars, / The disputants, I ween, / Rail on in utter ignorance / Of what each other mean, / And prate about an Elephant / Not one of them has seen!"[1] It is perhaps

1. John Godfrey Saxe, "The Blind Men and the Elephant." http://www.wordinfo.info/words/index/info/view_unit/1/?letter=B&spage=3.

Configurations, 2008, 16:299–324 © 2010 by The Johns Hopkins University Press and the Society for Literature and Science.

profane to compare a poem on the incomprehensibility of the divine to arguments over new media, but the invisibility, ubiquity, and alleged power of new media (and technology more generally) lend themselves nicely to this analogy. It seems impossible to know the extent, content, and effects of new media. Who knows the entire contents of the WWW or the real extent of the Internet or of mobile networks? How can one see and know all time-based, online interactions? Who can expertly move from analyzing social-networking sites to Japanese cell-phone novels to *World or Warcraft* to hardware algorithms to ephemeral art installations? Is a global picture of new media possible?

In response to these difficulties, an important strain of new media has moved away from content and from specific technologies to what seems to be common to all new media objects and moments: software. All new media allegedly rely on—or, most strongly, can be reduced to—software, a "visibly invisible" essence. Software seems to allow one to grasp the entire elephant because it is the invisible whole that generates the sensuous parts. Based on and yet exceeding our sense of touch—based on our ability to manipulate virtual objects we cannot entirely see—it is a magical source that promises to bring together the fractured field of new media studies and to encapsulate the difference this field makes.

But, what is software? What does it mean to know software and, most importantly, what does positing software as the essence of new media do? This essay responds to these questions, arguing that software as source relies on a profound logic of "sourcery"—a fetishism that obfuscates the vicissitudes of execution and makes our machines demonic. Further, this sourcery is the obverse rather than the opposite of the other dominant trend in new media studies: the valorization of the user as agent. These sourceries create a causal relationship among one's actions, one's code, and one's interface. The relationship among code and interface, action and result, however, is always contingent and always to some extent imagined. The reduction of computer to source code, combined with the belief that users run our computers, makes us vulnerable to fantastic tales of the power of computing. To break free of this sourcery, we need to interrogate, rather than venerate or even accept, the grounding or logic of software. Crucially, though, closely engaging software will not let us escape fictions and arrive at a true understanding of our machines, but rather make our interfaces more productively spectral. As a fetish, source code can provide surprising "deviant" pleasures that do not end where they should. Framed as a *re*-source, it can help us think through the machinic and human rituals that help us imagine our technologies and their executions.

The Logos of Software

To exaggerate slightly, software has recently been posited as the essence of new media, and knowing software as a form of enlightenment. Software is allegedly the truth, the base layer, the logic of new media. Lev Manovich in his groundbreaking *The Language of New Media* for instance asserts that

> new media may look like old media, but this is only the surface . . . to understand the logic of new media, we need to turn to computer science. It is there that we may expect to find the new terms, categories, and operations that characterize media that become programmable. *From media studies we move to something that can be called "software studies"—from media theory to software theory.*[2]

his turn to software—to the logic of what lies beneath—has offered a solid ground to new media studies, allowing it, as Manovich argues, to engage presently existing technologies and to banish so-called "vapor theory," theory that fails to distinguish between demo and product, fiction and reality, to the margins.[3] This call to banish vapor theory, made by Geert Lovink and Alexander Galloway amongst others, has been crucial to the rigorous study of new media, but, this rush away from what is vapory—undefined, set in motion—is also troubling because vaporiness is not accidental, but rather essential to, new media and, more broadly, to software. Software, after all, is ephemeral, information ghostly, and new media projects that have never, or barely, materialized are among the most valorized and cited. (Also, if you take the technical definition of information seriously, information increases with vapor, with entropy). This turn to computer science also threatens to reify knowing software as truth, an experience that is arguably impossible: we all know some software, some programming languages, but does anyone really "know" software? What could this knowing even mean? Regardless, from myths of all-powerful hackers who "speak the language of computers as one does a mother tongue"[4] or who produce abstractions that re-

2. Lev Manovich, *The Language of New Media* (Cambridge, MA: MIT Press, 2001), p. 48, emphasis in original.

3. Vapor theory is a term coined by Peter Lunenfeld and used by Geert Lovink to designate theory so removed from actual engagement with digital media that it treats fiction as fact. This term, however, can take on a more positive resonance, if on takes the non-materiality of software seriously. (Geert Lovink, "Enemy of Nostalgia, Victim of the Present, Critic of the Future Interview with Peter Lunenfeld," 31 Jul 2000 <http://www.nettime.org/Lists-Archives/nettime-l-0008/msg00008.html> accessed 2/1/2007).

4. Alexander R. Galloway, *Protocol: How Power Exists after Decentralization* (Cambridge, MA: MIT Press, 2004), p. 164.

lease the virtual[5] to perhaps more mundane claims made about the radicality of open source, knowing (or using) the right software has been made analogous to man's release from his self-incurred tutelage.[6] As advocates of free and open source software make clear, this critique aims at political, as well as epistemological, emancipation: as a form of enlightenment, it is a stance of how not to be governed like that—an assertion of an essential freedom that can only be curtailed at great cost.[7]

To be clear, I am not dismissing the political potential of free or open source software, or the importance of studying or engaging software; rather, I am arguing that we need to interrogate how knowing (or using free or open source) software does not simply enable us to fight domination or rescue software from evil-doers such as Microsoft, but rather is embedded in—mediates between, is part of—structures of knowledge-power. For instance, using free software does not mean escaping from power, but rather engaging it differently, for free and open source software profoundly privatizes the public domain: GNU *copyleft* does not seek to reform *copyright*, but rather to spread it everywhere.[8] It is thus symptomatic of the move in contemporary society away from the public/private

5. McKenzie Wark, "A Hacker Manifesto." version 4.0 (n.d.). http://subsol.c3.hu/subsol_2/contributors0/warktext.html.

6. See Richard Stallman, "The Free Software Movement and the Future of Freedom; March 9th 2006." http://fsfeurope.org/documents/rms-fs-2006-03-09.en.html. Immanuel Kant famously described enlightenment as "mankind's exit from its self-incurred immaturity" ("An Answer to the Question: What Is Enlightenment," *What Is Enlightenment? Eighteenth-Century Answers and Twentieth-Century Questions*, Ed. James Schmidt (Berkeley: California UP, 1996), 58).

7. For more on enlightenment as a stance of how not to be governed like that, see Michel Foucault, "What Is Critique?" in *What Is Enlightenment?,* ed. James Schmidt (Berkeley: University of California Press, 1996), pp. 382–398.

8. As Niva Elkin-Koren notes in "Creative Commons: A Skeptical View of a Worthy Project," the Creative Commons strategy "does not aim at creating a public domain, at least not in the strict legal sense of a regime that is free of any exclusive proprietary rights. The strategy is entirely dependent upon a proprietary regime and drives its legal force from its existence. The normative framework assumes that it is possible to replace existing practices of producing and distributing informational works by relying on the existing proprietary regime. The underlying assumption is that if intellectual-property rights remain the same, but rights are exercised differently by their owners, free culture would emerge" (http://www.hewlett.org/NR/rdonlyres/6D4BFD1E-09BB-4F89-9208-7C1E4B141F2A/0/Creative_Commons_Amsterdam_final2006.pdf). Although Elkin-Koren is writing about Creative Commons in this passage, she makes it clear that this strategy of extending and revising intellectual-property rights is drawn from the Free Software Movement's GPL.

dichotomy to that of open/closed.[9] More subtly, the free software movement, in insisting that freedom stems from free software—from freely accessible source code—amplifies the power of source code, erasing the vicissitudes of execution and the structures that ensure the coincidence of code and its execution. It buttresses the logic of software—that is, software as *logos*.

Software as we now know it (importantly, software was not always software) conflates word with result, logos with action. The goal of software is to conflate an event with a written command. Software blurs the difference among human-readable code (readable because of another program), its machine-readable interpretation, and its execution by turning the word "program" from a verb to a noun, by turning process in time into process in space, by turning execution into inscription—or at least attempting to do so. An example I've used elsewhere, Edsger Dijkstra's famous condemnation of "go to" statements, encapsulates this nicely.[10] In "Go to Statement Considered Harmful," Dijkstra argues, "the quality of programmers is a decreasing function of the density of go to statements in the programs they produce," because go to statements work against the fundamental tenant of what Dijkstra considered to be good programming: namely, the necessity to "shorten the conceptual gap between static program and dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible."[11]

This is important, since if a program suddenly halts because of a bug, go tos make it difficult to find the place in the program that corresponds to the buggy code. Go tos make difficult the conflation of instruction with its product—the reduction of process to command—that grounds the emergence of software as a concrete entity and commodity; that is, go tos make it difficult for the source program to act as a legible source.[12] As I've argued elsewhere, this conflation of

9. For more on this, see Wendy Hui Kyong Chun, *Control and Freedom: Power and Paranoia in the Age of Fiber Optics* (Cambridge, MA: MIT Press, 2006).

10. See Wendy Hui Kyong Chun, "On Software, or the Persistence of Visual Knowledge," *grey room* 18 (winter 2005): 27–52.

11. Edsger Dijkstra, "Go to Statement Considered Harmful," in *Software Pioneers: Contributions to Software Engineering*, ed. Manfred Broy and Ernst Denert (Berlin: Springer, 2002), p. 342.

12. Structured programming was introduced as a way to make programs, rather than "programmer priest," the source, although the term programmer priest complicates the notion of source: Is the source the programmer, or is it some mythic power she mediates?

instruction or command with its product is also linked to software's gendered, military history: in the military, there is supposed be no difference between a command given and a command completed, especially to a "girl."[13] The implication here is: execution does not matter—as in conceptual art, execution is a perfunctory affair; what really matters is the source.[14]

This drive to forget or trivialize execution is not limited to programmers confounded with the task of debugging errant programs; it also extends to many critical analyses of code. These theorizations, which importantly question the reduction of new media with screen, emphasize code as performative or executable. For instance, Alexander Galloway, in *Protocol: How Control Exists after Decentralization,* claims: "code draws a line between what is material and what is *active*, in essence saying that writing (hardware) cannot *do* anything, but must be transformed into code (software) to be effective. . . . Code is language, but a very special kind of language. *Code is the only language that is executable*."[15] Drawing in part from Galloway, N. Katherine Hayles, in *My Mother Was a Computer: Digital Subjects and Literary Texts,* distinguishes between the linguistic performative and the machinic performative by arguing that

> [c]ode that runs on a machine is performative in a much stronger sense than that attributed to language. When language is said to be performative, the kinds of actions it "performs" happen in the minds of humans, as when someone says, "I declare this legislative session open" or "I pronounce you husband and wife." Granted, these changes in minds can and do reach in behavioral effects, but the performative force of language is nonetheless tied to external changes through complex chains of mediation. Code running in a digital computer causes changes in machine behavior and, through networked ports and other interfaces, may initiate other changes, all implemented through transmission and execution of code.[16]

13. See Chun, "On Software" (above, n. 11).

14. For more on software art as conceptual art, see Florian Cramer, "Concepts, Notations, Software, Art." 2002. http://userpage.fu-berlin.de/~cantsin/homepage/writings/software_art/concept_notations/concepts_notations_software_art.html.

15. Galloway, *Protocol* (above, n. 5), p. 165, emphasis in original.

16. N. Katherine Hayles, *My Mother Was a Computer: Digital Subjects and Literary Texts* (Chicago: University of Chicago Press, 2005), p. 50. Hayles's argument immediately poses the question: What counts as internal versus external to the machine, especially given that, in John von Neumann's foundational description of stored program computing, the input and output (the outside world to the machine) was a form of memory?

The independence of machine action—this autonomy, or automatic executability of code—is, according to Galloway, its material essence:

> the material substrate of code, which must always exist as an amalgam of electrical signals and logical operations in silicon, however large or small, demonstrates that code exists first and foremost as commands issued to a machine. Code essentially has no other reason for being that instructing some machine in how to act. One cannot say the same for the natural languages.[17]

Galloway thus strongly states, in "Language Wants to Be Overlooked: Software and Ideology," that "to see code as subjectively performative or enunciative is to anthropomorphize it, to project it onto the rubric of psychology, rather than to understand it through its own logic of 'calculation' or 'command.'"[18]

To what extent, however, can source code be understood outside of anthropomorphization? Does understanding voltages stored in memory as commands/code not already anthropomorphize the machine? (The inevitability of this anthropormorphization is arguably evident in the title of Galloway's article: "Language *Wants* to Be Overlooked" [emphasis added].) How is it that code "causes" changes in machine behavior? What mediations sustain the notion of code as inherently executable?

To make the argument that code is automatically executable, the process of execution itself must not only be erased, but source code also must be conflated with its executable version. This is possible because, it is argued, the two "layers" of code can be reduced to each other. Indeed, in *Protocol*, Galloway argues that

> uncompiled source code is *logically* equivalent to the same code compiled into assembly language and/or linked into machine code. For example, it is absurd to claim that a certain value expressed as a hexadecimal (base 16) number is more or less fundamental than that same value expressed as [a] binary (base 2) number. They are simply two expressions of the same value.[19]

He later elaborates on this point by drawing an analogy between quadratic equations and logical layers:

> One should never understand this "higher" symbolic machine as anything empirically different from the "lower" symbolic interactions of voltages

17. Alexander R. Galloway, "Language Wants to Be Overlooked: Software and Ideology," *Journal of Visual Culture* 5:3 (2006): 315–331.

18. Ibid., 321.

19. Galloway, *Protocol* (above, n. 5), p. 167, emphasis in original.

> through logic gates. They are complex aggregates yes, but it is foolish to think that writing an "if/then" control structure in eight lines of assembly code is any more or less machinic than doing it in one line of C, just as the same quadratic equation may swell with any number of multipliers and still remain balanced. The relationship between the two is *technical*.[20]

According to Galloway's quadratic equation analogy, the difference between a compact line of higher-level programming code and eight lines of assembler equals the difference between two equations, in which one contains coefficients that are multiples of the other. The solution to both equations is the same: one equation can be reduced to the other. This reduction, however, does not capture the difference between the various instantiations of code, let alone the empirical difference between the higher symbolic machine and the lower interactions of voltages (the question here is: Where does one make the empirical observation?); that is, to push Galloway's conception further, a technical relation is far more complex than a numerical one.

Significantly, a technical relation engages art or craft. According to the *Oxford English Dictionary* (*OED*), a technical person is one "skilled in or practically conversant with some particular art or subject." Code does not always nor automatically do what it says, but does so in a crafty manner. To state the obvious, one cannot run source code: it must be compiled or interpreted. This compilation or interpretation—this making code executable—is not a trivial action; the compilation of code is not the same as translating a decimal number into a binary one; rather, it involves instruction explosion and the translation of symbolic into real addresses. Consider, for example, the instructions needed for adding two numbers in PowerPC assembly language, which is one level higher than machine language:

```
li      r3,1          *load the number 1 into register 3
li      r4,2          *load the number 2 into register 4
add     r5,r4,r3      *add r3 to r4 and store the result in r5
stw     r5,sum(rtoc)  *store the contents of r5 (i.e. 3)into the
                        memory location
                      *called "sum"s (where sum is defined elsewhere)
blr                   *end of this snippet of code[21]
```

20. Galloway, "Language Wants to Be Overlooked" (above, n. 18), p. 321, emphasis in original.

21. This example draws from the *PowerPC Assembly Language Beginners Guide*. http://www.lightsoft.co.uk/Fantasm/Beginners/Chapt1.html.

This explosion is not equivalent to multiplying both sides of a quadratic equation by the same coefficient or to the difference between E and 15. It is, rather, a breakdown of the steps needed to perform what seems a simple arithmetic calculation. It is the difference between a mathematical identity and a logical equivalence, which depends on a leap of faith. This is most clear in the use of numerical methods to turn integration—a function performed fluidly in analog computers—into a series of simpler, repetitive arithmetical steps. This translation from source code to executable is arguably as involved as the execution of any command. More importantly, it depends on the action (human or otherwise) of compiling/interpreting and executing. Also, some programs may be executable, but not all compiled code within that program is executed; rather, lines are read in as necessary. *So, source code thus only becomes source after the fact.* Source code is more accurately a *re-source,* rather than a source. Source code becomes a source when it becomes integrated with logic gates (and at an even lower level, with the transistors that comprise these gates); when it expands to include software libraries, when it merges with code burned into silicon chips; and when all these signals are carefully monitored, timed, and rectified.

Code is also not always the source, because hardware does not need software to "do something." One can build algorithms using hardware, for all hardware cannot be reduced to stored program memory. Figure 1, for instance, is the logical statement: "if notB and notA, do CMD1 (state P); if notB and notA and notZ OR B and A (state Q) then command 2." To be clear, this is not a valorization of hardware over software, as though hardware necessarily escapes this drive to conflate space with time. Crucially, this schematic is itself an abstraction. Logic gates can only operate "logically" if they are carefully timed. As Philip Agre has emphasized, the digital abstraction erases the fact that gates have "directionality in both space (listening to its inputs, driving its outputs) and in time (always moving toward a logically consistent relation between these inputs and outputs)."[22] When a value suddenly changes, there is a brief period in which a gate will give a false value. In addition, because signals propagate in time over space, they produce a magnetic field that can corrupt other signals nearby ("crosstalk"). This schematic erases all these various time-based effects. Thus hardware schematics, rather than escaping from the logic of sourcery, are also embedded within this structure. Indeed, John von Neumann, the alleged

22. Philip E. Agre, *Computation and Human Experience* (Cambridge: Cambridge University Press, 1997), p. 92.

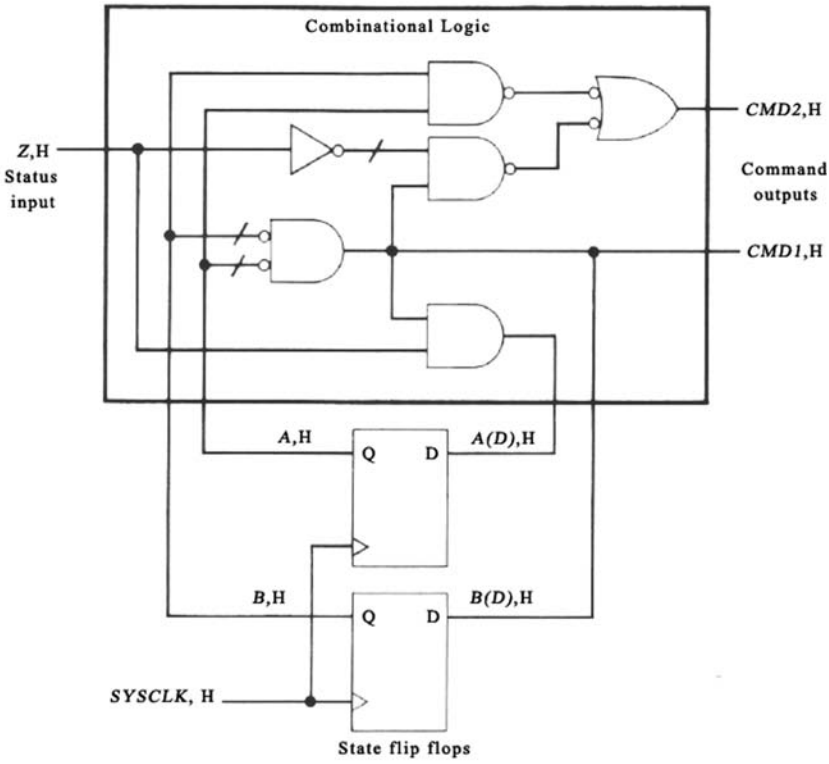

Figure 1. If notB and notA, do CMD1 (state P); if notB and notA and notZ OR B and A (state Q) then command.

architect of the stored-memory digital computer, drew from Warren McCulloch and Walter Pitts's conflation of neuronal activity with its inscription in order to conceptualize our modern computers. According to McCulloch and Pitts (who themselves drew from Alan Turing's work), "the response of any neuron [is] factually equivalent to a proposition which proposed its adequate stimulus."[23] That is, the response of a neuron can be reduced to the circumstances that make it possible: instruction can substitute for result. It is perhaps appropriate then that von Neumann spent the last days of his life—dying from a cancer most probably stemming from his work at Los Alamos—reciting from memory *Faust,* Part 1.[24] At the heart

23. Warren McCulloch and Walter Pitts, "A Logical Calculus of the Ideas Immanent in Nervous Activity," in *Embodiments of Mind* (Cambridge, MA: MIT Press, 1965), p. 21.

24. Norman McRae, *John von Neumann* (New York: Pantheon, 1992), p. 378.

of stored-program computing lies the Faustian substitution of word for action.

Not surprisingly, this notion of source code as source coincides with the introduction of alpha-numeric languages. With them, human-written, nonexecutable code becomes source code and the compiled code, the object code. Source code thus is arguably symptomatic of human language's tendency to attribute a sovereign source to an action, a subject to a verb.[25] By converting action into language, source code emerges. Thus Galloway's statement "to see code as subjectively performative or enunciative is to anthropomorphize it, to project it onto the rubric of psychology, rather than to understand it through its own logic of 'calculation' or 'command'" overlooks the fact that to use higher-level alphanumeric languages is already to anthropomorphize the machine, and to reduce all machinic actions to the commands that supposedly drive them. In other words, the fact that "code is law"—something Lawrence Lessig pronounces with great aplomb—is hardly profound.[26] After all, code is, according to the *OED*, "a systematic collection or digest of the laws of a country, or of those relating to a particular subject." What is surprising is the fact that *software is code*, that code is—has been made to be—executable, and that this executability makes code not law, but rather every lawyer's dream of what law should be, automatically enabling and disabling certain actions and functioning at the level of everyday practice.

Source Code as Fetish

Source code as source means that software functions as an axiom, as "a self-evident proposition requiring no formal demonstration to prove its truth, but received and assented to as soon as it is mentioned."[27] In other words, whether or not source code is only source code after the fact or whether or not software can be physically separated from hardware,[28] software is always posited as already existing, as the self-evident ground or source of our interfaces. Software is axiomatic. As a first principle, it fastens in place a certain

25. According to Friedrich Nietzsche in *The Genealogy of Morals*, "there is no 'being' behind the doing, effecting, becoming: 'the doer' is merely a fiction added to the deed—the deed is everything"; see Nietzsche, *The Birth of Tragedy & Genealogy of Morals*, trans. Francis Golffing (New York: Doubleday, 1956), pp. 178–179.

26. Lawrence Lessig, *Code and Other Laws of Cyberspace* (New York: Basic Books, 1999).

27. *Oxford English Dictionary Online*. http://www.oed.com.

28. See Friedrich Kittler, "There Is No Software," *Ctheory*. 1995. http://www.ctheory.net/articles.aspx?id=74.

logic of cause and effect, based on the erasure of execution and the privileging of programming that bleeds elsewhere and stems from elsewhere as well.[29] As an axiomatic, it, as Gilles Deleuze and Félix Guattari argue, artificially limits decodings.[30] It temporarily limits what can be decoded, put into motion, by setting up an artificial limit—the artificial limit of programmability—that seeks to separate information from entropy by designating some entropy information and other "nonintentional" entropy noise. Programmability, discrete computation, depends on the disciplining of hardware and the desire for a programmable axiomatic. Code is a medium in the full sense of the word. As a medium, it channels the ghost that we imagine runs the machine—that we see as we don't see—when we gaze at our screen's ghostly images.

Understood this way, source code is a fetish. According to the *OED*, a fetish was originally an ornament or charm worshipped by "primitive peoples . . . on account of its supposed inherent magical powers."[31] The term *fetisso* stemmed from the trade of small wares and magic charms between the Portuguese merchants and West Africans; Charles de Brosses, in 1757, coined the term fetishism to describe "primitive religions." According to William Pietz, Enlightenment thinkers viewed fetishism as a

> false causal reasoning about physical nature [that became] the definitive mistake of the pre-enlightened mind: it superstitiously attributed intentional purpose and desire to material entities of the natural world, while allowing social action to be determined by the . . . wills of contingently personified things, which were, in truth, merely the externalized material sites fixing people's own capricious libidinal imaginings.[32]

That is, fetishism, as "primitive causal thinking," derived causality from desire rather than reason:

> Failing to distinguish the intentionless natural world known to scientific reason and motivated by practical material concerns, the savage (so it was argued) superstitiously assumed the existence of a unified causal field for personal actions and physical events, thereby positing reality as subject to animate powers whose purposes could be divined and influenced. Specifically, humanity's

29. Namely, twentieth-century genetics.

30. Gilles Deleuze and Félix Guattari, "Capitalism: A Very Special Delirium." 1995. http://www.generation-online.org/p/fpdeleuze7.htm.

31. *Oxford English Dictionary Online* (above, n. 28).

32. William Pietz, "Fetishism and Materialism," in *Fetishism as Cultural Discourse*, ed. Emily Apter and William Pietz (Ithaca, NY: Cornell University Press, 1993), pp. 138, 139.

> belief in gods and supernatural powers (that is, humanity's unenlightenment) was theorized in terms of prescientific peoples' substitution of imaginary personifications for the unknown physical causes of future events over which people had no control and which they regarded with fear and anxiety.[33]

Fetishes thus allow the human mind too much and not enough control by establishing a "unified causal field" that encompasses both personal actions and physical events. Fetishes enable a semblance of control over future events—a possibility of influence, if not an airtight programmability—that itself relies on distorting real social relations into material givens.

This notion of fetish as false causality has been most important to Karl Marx's diagnosis of capital as fetish. He famously argued that

> the commodity-form . . . is nothing but the determined social relation between humans themselves which assumes here, for them, the phantasmagoric form of a relation between things. In order, therefore, to find an analogy we must take a flight into the misty realm of religion. There the products of the human head appear as autonomous figures endowed with a life of their own, which enter into relations both with each other and with the human race. So it is in the world of commodities with the products of men's hands. This I call . . . fetishism.[34]

The capitalist thus confuses social relations and the labor activities of real individuals with capital and its seeming magical ability to reproduce, for "it is in interest-bearing capital . . . that capital finds its most objectified form, its pure fetish form. . . . Capital—as an entity—appears here as an independent source of value; a something that creates value in the same way as land [produces] rent, and labor wages."[35]

The parallel to source code seems obvious: we "primitive folk" worship source code as a magical entity—as a source of causality—when in truth the power lies elsewhere, most importantly in social and machinic relations. If code is performative, its effectiveness relies on human and machinic rituals. Intriguingly though, in this parallel, Enlightenment thinking—a belief that knowing leads to control and freedom, a release from tutelage—is not the "solution" to the fetish, but rather what grounds it, for source code historically has been portrayed as the solution to wizards and other myths of

33. Ibid., p. 137.

34. Karl Marx, *Capital: A Critique of Political Economy*, vol. 1, trans. Ben Foskes (New York: Penguin, 1976), p. 165.

35. Marx, as quoted by Pietz, "Fetishism and Materialism" (above, n. 33), p. 149.

programming. According to this popular narrative, machine code provokes mystery and submission; source code enables understanding and thus institutes rational thought and freedom, enabling us to move from being blind to sighted.

John Backus, writing about programming in the 1950s, contends that "programming in the early 1950s was a black art, a private arcane matter."[36] These programmers formed a "priesthood guarding skills and mysteries far too complex for ordinary mortals."[37] Opposing even the use of decimal numbers, these machine programmers were sometimes deliberate purveyors of their own fetishes or "snake oil," systems that allegedly had "mysterious, almost human abilities to understand the language and needs of the user; closer inspection was likely to reveal a complex, exception-ridden performer of tedious clerical tasks that substituted its own idiosyncrasies for those of the computer."[38] Automatic programming languages such as FORTRAN, developed by Backus, allegedly exorcised the profession of the priesthood by making programs more readable and thus making it easier to discern the difference between snake oil and the real thing.

Similarly, Richard Stallman, in his critique of nonfree software, has argued that machine-executable programs create mystery rather than enlightenment. Stallman contends that an executable "is a mysterious bunch of numbers. What it does is secret."[39] As I argue below, this notion of executables and executing code as magic is buttressed by real-time operating systems that posit the user as the source. Against this magical execution, source code supposedly enables pure understanding and freedom—the ability to map and understand the workings of the machine, but again only through a magical erasure of the gap between source and execution, an erasure of execution itself. Tellingly, this move to source code has hardly deprived programmers of their priest-like/wizard status. If anything, the notion of programmers as super-human has been disseminated ever more, and the history of computing—from direct manipulation to hypertext—has been marked by various "liberations."

But clearly, source code can do things: it can be interpreted or compiled, and it can be rendered into machine-readable commands

36. John Backus, "Programming in America in the 1950s—Some Personal Impressions," in *A History of Computing in the Twentieth Century*, ed. N. Metropolis et al. (New York: Academic Press, 1980), p. 126.

37. Ibid., p. 127.

38. Ibid.

39. Richard Stallman, "Copyright and Globalization in the Age of Computer Networks." 2001. http://www.gnu.org/philosophy/copyright-and-globalization.html.

that are then executed. Source code is also read by humans and is written by humans for humans: it is thus the source of some understanding. Although Ellen Ullman and many others have argued that "a computer program has only one meaning: what it does. It isn't a text for an academic to read. Its entire meaning is its function," source code must be able to function, even if it does not function—that is, even if it is never executed.[40] Source code's readability is not simply due to comments that are embedded in the code, but also due to English-based commands and programming styles designed for comprehensibility. This readability is not just for "other programmers"; when programming, one must be able to read one's own program—to follow its logic and predict its outcome, whether or not this outcome coincides with one's prediction.

This notion of source code as readable—as creating some outcome regardless of its machinic execution—underlies "codework" and other creative projects. The Internet artist Mez, for instance, has created a language, "mezangelle," that incorporates formal code and informal speech. Mez's poetry deliberately plays with programming syntax, producing language that cannot be executed but nonetheless draws on the conventions of programming language to signify.[41] Codework, however, can also work entirely within an existing programming language. Graham Harwood's perl poem, for example, translates William Blake's late-eighteenth-century poem "London" into London.pl, a script that contains within it an algorithm to "find and calculate the gross lung-capacity of the children screaming from 1792 to the present."[42] Regardless of whether or not it can execute, code can be, must be, worked into something meaningful. Source code, in other words, may be sources of things other than the machine execution it is "supposed" to engender.

Source code as fetish, understood psychoanalytically, embraces this nonteleological potential of source code, for the fetish is a genital substitute that gives the fetishist nonreproductive pleasure. It is a deviation that does not "end" where it should, a deviation taken on so that the child may combat castration for both himself and his mother, while at the same time accommodating to his world's larger Oedipal structure. It both represses and acknowledges paternal symbolic authority. According to Freud, "the fetish is a substitute for the woman's (mother's) phallus which the little boy once believed in

40. Ellen Ullman, interviewed by Scott Rosenberg, "21st: Elegance and Entropy." 1997. http://dir.salon.com/story/tech/feature/1997/10/09/interview/print.html.

41. See mez's site at http://www.hotkey.net.au/~netwurker/.

42. See http://www.scotoma.org/notes/index.cgi?LondonPL.

and does not wish to forego,"[43] but the fetish, formed the moment the little boy sees his mother's phallus, also transforms the phallus:

> It is not true that the child emerges from his experience of seeing the female parts with an unchanged belief in the woman having a phallus. He retains this belief but he also gives it up; during the conflict between the deadweight of the unwelcome perception and the force of the opposite wish, a compromise is constructed such as is only possible in the realm of unconscious thought—by the primary processes. In the world of psychical reality the woman still has a penis in spite of it all, but this penis is no longer the same as it once was. Something else has taken its place, has been appointed as its successor, so to speak, and now absorbs all the interest which formerly belonged to the penis.[44]

The fetish is a transformation of the mother's phallus, whose magical power "remains a token of triumph over the threat of castration and a safeguard against it."[45] As such, it both fixes a singular event—turning time into space—and enables a logic of repetition that constantly enables this safeguarding. As Pietz argues

> the fetish is always a meaningful fixation of a singular event; it is above all a "historical" object, the enduring material form and force of an unrepeatable event. This object is "territorialized" in material space (an earthly matrix), whether in the form of a geographical locality, a marked site on the surface of the human body, or a medium of inscription or configuration defined by some portable or wearable thing.[46]

Although it fixes a singular event, the fetish works only because it can be repeated, but again, what is repeated is both denial and acknowledgment, since the fetish can be "the vehicle both of denying and asseverating the fact of castration."[47]

Slavoj Zizek draws on this insight to explain the persistence of the Marxist fetish:

> When individuals use money, they know very well that there is nothing magical about it—that money, in its materiality, is simply an expression of social relations. . . . on an everyday level, the individuals know very well that there are relations between people behind the relations between things. The problem is that in their social activity itself, in what they are *doing*, they are *acting*

43. Sigmund Freud, "Fetishism," in *Sexuality and the Psychology of Love* (New York: Macmillan, 1963), p. 205.

44. Ibid., p. 206.

45. Ibid.

46. William Pietz, "The Problem of the Fetish," pt. 1. *Res* 9 (spring 1985): 12.

47. Freud, "Fetishism" (above, n. 44), p. 208.

> as if money, in its material reality is the immediate embodiment of wealth as such. They are fetishists in practice, not in theory. What they "do not know," what they misrecognize, is the fact that in their social reality itself—in the act of commodity exchange—they are guided by the fetishistic illusion.[48]

Fetishists importantly know what they are doing—knowledge is not an answer to fetishism. The knowledge that source code offers is therefore no cure for source-code fetishism: if anything, this knowledge sustains it.

To make explicit the parallels, source code, like the fetish, is a conversion of event into location—time into space—that does affect things, but not necessarily in the manner prescribed. Its effects can be both productive and nonexecutable. Also, in terms of denial and acknowledgment, we know very well that source code is not executable, yet we persist in treating it as so. And it is this glossing over that makes possible the ideological belief in programmability.

Code as fetish means that computer execution deviates from the so-called source, as source program does from programmer. Alan Turing, in response to the objection that computers cannot think because they merely follow human instructions, argued:

> Machines take me by surprise with great frequency. . . . The view that machines cannot give rise to surprises is due, I believe, to a fallacy to which philosophers and mathematicians are particularly subject. This is the assumption that as soon as a fact is presented to a mind, all consequences of that fact spring into the mind simultaneously with it. It is a very useful assumption under many circumstances, but one too easily forgets that it is false. A natural consequence of doing so is that one then assumes that there is no virtue in the mere working out of consequences from data and general principles.[49]

This erasure of the vicissitudes of execution coincides with the conflation of data with information, information with knowledge—the assumption that what's most difficult is the capture, rather than the analysis, of data. This erasure of execution through source code as source creates an intentional authorial subject: the computer, the program, or the user, and this source is treated as the source of meaning. The fact that there is an algorithm, a meaning intended by code (and thus in some way knowable), sometimes structures our experience with programs. When we play a game, we arguably try to reverse-engineer its algorithm, or at the very least to link

48. Slavoj Zizek, *The Sublime Object of Ideology* (New York: Verso, 1989), p. 31, emphasis in original.

49. Alan Turing, "Computing Machinery and Intelligence," *Mind* 59. 1950. http://www.loebner.net/Prizef/TuringArticle.html.

its actions to its programming, which is why all design books warn against coincidence or random mapping, since it can induce paranoia in its users. That is, because an interface is programmed, most users treat coincidence as meaningful. To the user, as with the paranoid schizophrenic, there is always meaning: whether or not the user knows the meaning, she or he knows that it regards her or him. Whether or not she or he cares, however, is another matter. Regardless, the fact that the code allegedly driving an interface can be revealed, the fact that source code exists means that the truth is out there. To know the code is to have a form of "X-ray" vision that makes the inside and outside coincide, and the act of revealing sources or connections becomes a critical act in and of itself.[50]

Real Time, This Time

This discussion of the user brings to light an intriguing fact: the notion of user as source, introduced though real-time operating systems, is the obverse, rather than the antidote, to code as source. According to the *OED*, real time is "the actual time during which a process or event occurs, especially one analyzed by a computer, in contrast to time subsequent to it when computer processing may be done, a recording replayed, or the like." Crucially, hard and soft real-time systems are subject to a "real-time constraint"—that is, they need to respond, in a forced duration, to actions predefined as events. The measure of real time, in computer systems, is its reaction to the live—its liveness.

The notion of real time always points elsewhere—to "real-world" events, to user's actions—thereby introducing indexicality to this supposedly nonindexical medium; that is, whether or not digital images are supposed to be "real," real time posits the existence of a source—coded or not—that renders our computers transparent. Real-time operating systems create an "abstraction layer" that hides the hardware details of the processor from application software; real-time images portray computers as un-mediated connectivity. As Realplayer reveals, the notion of real time is bleeding into all electronic moving images, not because all recordings are live, but because grainy moving images have become a marker of the real.[51] What is

50. N. Katherine Hayles develops this theme of revealing codes in *My Mother Was a Computer* (above, n. 17), pp. 54–61. Importantly, some software art projects also complicate and frustrate code as X-ray vision and connection as meaning, such as Golan Levin's *Axis Aplet*.

51. See Thomas Levin, "Rhetoric of the Temporal Index: Surveillant Narration and the Cinema of 'Real Time,'" *CTRL Space: Rhetorics of Surveillance from Bentham to Big Brother*, ed. Thomas Levin et al. (Cambridge, MA: MIT Press, 2002), pp. 578–593.

authentic or real is what transpires in real time, but real time is real not only because of this indexicality—this pointing to elsewhere—but also because its quick reactions to user's inputs.

Dynamic changes to webpages in real time, seemingly at the bequest of users' desires or inputs, create what Tara McPherson has called "volitional mobility." Creating "Tara's phenomenology of websurfing," McPherson argues:

> When I explore the web, I follow the cursor, a tangible sign of presence: implying movement. This motion structures a sense of liveness, immediacy, of the now . . . yet this is not the old liveness of television: this is liveness with a difference. This liveness foregrounds volition and mobility, creating a liveness on demand. Thus, unlike television which parades its presence before us, the web structures *a sense of causality* in relation to liveness, a liveness we navigate and move through, often structuring a feeling that our own desire drives the movement. The web is about presence but an unstable presence: it's in process, in motion.[52]

This liveness, McPherson carefully notes, is more the illusion—the feel or sensation—of liveness, rather than the fact of liveness; the choice yoked to this liveness is similarly a sensation rather than the real thing (although one might ask: What is the difference between the feel of choice and choice? Is choice itself not a limitation of agency?). The real-time moving cursor and the unfolding of an unstable present through our digital (finger) manipulations make us crane our necks forward, rather than sit back on our couches, causing back and neck pain. The extent to which computers turn the most boring activities into incredibly time-consuming and even enjoyable ones is remarkable: one of the most popular computer game to date, *The Sims*, focuses on the mundane; action and adventure games reduce adventure to formulaic, motion-restricted activities. This volitional mobility, McPherson argues, reveals that the "hype" surrounding the Internet does have some phenomenological backing. This does not necessarily make the Internet an empowering medium, but at the very least means that it can provoke a desire for something better: true volitional mobility, true change.[53]

52. Tara McPherson, "Reload: Liveness, Mobility and the Web," in *The Visual Culture Reader*, 2nd ed., ed. Nicholas Mirzoeff (New York: Routledge, 2002), p. 462, emphasis in original.

53. Coming from film rather than television studies and focusing more on applications than phenomenology, Galloway in *Protocol* (above, n. 5) similarly argues that continuity makes websurfing "a compelling, intuitive experience for the user": "On the Web, the browser's movement is experienced as the user's movement. The mouse movement is substituted for the user's movement. The user looks through the screen into an imaginary world, and it makes sense. The act of "surfing the web," which, phenomenologically,

As noted earlier, however, the user is not the only source of change in real time. Real-time images, such as those provided by webcams, make our computer screen seem, for however brief a moment, alive. Refreshing on their own and pointing elsewhere, they make our networks seem transparent and thus fulfill the promise of fiber-optic networks to connect us to the world, as do real-time stock quotes and running news banners.[54] As a whole, these moments of "interactivity" buttress the notion of transparency at a larger level. The source of a computer's actions always stems from elsewhere, because real time makes it appear as though only outside events—user mouse-clicks, streaming video—cause the computer actions. These real-time interactions, which were initially introduced to make computation more efficient, have almost erased computation altogether. Once again, the movements within the computer—the constant regeneration, the difference between the textual representation of a program, a compiled program, a program stored on the hard drive, and the program read-in instruction by instruction into the processor—are all erased. These movements create our spectral interface, without which our machines (we erroneously believe) could not work, and it is these spectral interfaces that allow us to include computers within the broader field of "visual-culture" studies, or screen media. Viewed as the alpha and omega of our computers, interfaces stand in, more often than not, for the computer itself, erasing the medium as it proliferates its specters, making our machines transparent producers of unreal visions—sometimes terrifying, but usually banal imitations or hallucinations of elsewhere.

Demonic Media

Not accidentally, this spectrality of digital media makes our media demonic; that is, inhabited by invisible processes that, perhaps like Socrates' *daimonion* (mystical inner voice), help us in our time of need. They make executables magic. UNIX—that operating system seemingly behind our happy spectral Macs—runs daemons. Daemons run our e-mail, our webservers. Macs thus not only proudly display that symbol of Judeo-Christian man's seduction and fall from grace—that sanitized but nonetheless telling bitten apple—it also inhabits its operating systems with daemons that make it a veritable "paradise lost" (see Fig. 2).

should be an unnerving experience of radical dislocation—passing from a server in one city to a server in another city—could not be more pleasurable for the user. Legions of computer users live and play online with no sense of radical dislocation" (p. 64).

54. For more, see chapter 5 of Chun's *Control and Freedom* (above, n. 10).

Figure 2. FreeBSD mascot.

So why are these daemons called "send mail" and not Satan? Most simply, a daemon is a process that runs in the background without intervention by the user (usually initiated at boot time). They can run continuously, or in response to a particular event or condition (for instance, network traffic), or at a scheduled time (e.g., every five minutes, or at 05:00 every day). More technically, UNIX daemons are parentless—that is, orphaned—processes that run in the root directory. You can create a UNIX demon by forking a child process and then having the parent process exit, so that INIT (the daemons of daemons) takes over as the parent process.[55]

55. The following PERL program, for instance, says hello every five minutes (from http://www.webreference.com/perl/tutorial/9/3.html):

```
use POSIX qw(setsid);
#turns the process into a session leader, group leader, and ensures that it doesn't
#have a controlling terminal
chdir '/' or die "Can't chdir to /: $!";
umask 0;
open STDIN, '/dev/null' or die "Can't read /dev/null: $!";
#open STDOUT, '>/dev/null' or die "Can't write to /dev/null: $!";
open STDERR, '>/dev/null' or die "Can't write to /dev/null: $!";
defined(my $pid = fork) or die "Can't fork: $!";
exit if $pid;
setsid or die "Can't start a new session: $!";
while(1) {
sleep(5);
print "Hello...\n";
}
```

UNIX daemons supposedly stem from the Greek word *daemon,* meaning, according to the *OED*, "a supernatural being of a nature intermediate between that of gods and men; an inferior divinity, spirit, genius (including the souls or ghosts of deceased persons, esp. deified heroes)." A daemon is thus already a medium, an intermediate value, albeit one that is not often seen. The most famous daemon is perhaps Socrates' *daimonion*—that mystical inner voice that assisted Socrates in time of crisis by forbidding him to do something rash. The other famous daemon, more directly related to those spawning UNIX processes, is Maxwell's demon. According to Fernando Corbato, one of the original members of the Project MAC group in 1963:

> Our use of the word daemon was inspired by the Maxwell's daemon of physics and thermodynamics. (My background is Physics.) Maxwell's daemon was an imaginary agent which helped sort molecules of different speeds and worked tirelessly in the background. We fancifully began to use the word daemon to describe background processes which worked tirelessly to perform system chores.[56]

Daemonic processes are slaves that work tirelessly and, like all slaves, define and challenge the position of the master.

The introduction of multiuser, command line processing—real-time operating systems—necessitates the mystification of processes that seem to operate automatically without user input, breaking the interfaces' "diegesis." What is *not seen* becomes daemonic rather than what is normal, because the user is supposed to be the cause and end of any process. Real-time operating systems, such as UNIX, transform the computer from a machine run by human operators in batch-mode to "alive" personal machines, which respond to users' commands. Real-time content—stock quotes, breaking news, and streaming video—similarly transforms personal computers into personal media machines. These moments of "interactivity" buttress the notion of our computers as transparent. Real-time processes, in other words, make the user the "source" of the action, but only by orphaning those processes without which there could be no user. By making the interface transparent or "rational," one creates demons.

It is not perhaps surprising then that Nietzsche condemned Socrates so roundly for his daemon (and similarly language of its attribution of subject to verb). According to Nietzsche, Socrates was

56. "The Origin of the word Daemon." http://ei.cs.vt.edu/~history/Daemon.html. This is why Neal Stephenson, in *Snow Crash* (New York: Bantam, 1992), describes robots or servants in the *Metaverse* as daemons.

himself a daemon because he insisted on the transparency of knowledge, because he insisted that what is most beautiful is also most sensible. Crucially, Socrates' divine inner voice only spoke to dissuade. Socrates introduced order and reified conscious perception, making instinct the critic and consciousness the creator. Perhaps as a sign of the desire for the transparency of knowledge—the reigning of rationality—daemon is also a backronym. Since the first daemon was apparently a program that automatically made tape backups of the file system, it has been assumed that daemon stands for "Disk And Executive MONitor." This first daemon is appropriately about memory: an automated process, stored in memory, that transfers data between secondary and tertiary forms of memory, and that stores the code so that it can be viewed as source. Memory is what makes possible daemons, makes our media daemonic.[57] The questions that remain are: How to deal with these daemons and their alleged source codes? Should these daemons be exorcised, or is this spectral relationship not central to the very ghostly concept of information and the commodity itself?[58]

Code as Re-Source

To answer these questions, let me return to *code as re-source*, for code as re-source enables us to think in terms of the gap between source and execution, and makes an interface a process rather than a stable thing. This gap complicates any analysis of user determination by software: as Matthew Fuller points out, the "gap between a model of a function and its actualities . . . in some cases describes a degree of freedom, and . . . in others puts into place a paralyzing incapacity to act."[59] Reading Microsoft Word's mammoth feature mountain, Fuller compelling argues that the more features offered in an anxious attempt to program the user—the more codes provided—the more ways the user can go astray.[60] Thinking in terms of this gap also means thinking of how information is undead; that is, how it returns over and over again.[61] Second, code as re-source allows us to take seriously

57. See Wendy Chun, "The Enduring Ephemeral, or the Future Is a Memory," *Critical Inquiry* 35:1 (2008): 148–171.

58. See Thomas Keenan, "The Point Is the (Ex)Change It: Reading *Capital*, Rhetorically," in *Fetishism as Cultural Discourse*, ed. Emily Apter and William Pietz (Ithaca, NY: Cornell University Press, 1993), pp. 152–185.

59. Matthew Fuller, *Behind the Blip* (New York: Autonomedia, 2003), p. 107.

60. See Matthew Fuller, "It looks as though you're writing a letter," *Telepolis*. 2001. http://www.heise.de/tp/r4/artikel/7/7073/1.html.

61. See Chun, "Enduring Ephemeral" (above, n. 58).

the entropy, noise, and decay that code as source renders invisible. By taking decay seriously, we can move away from the conflation of storage with memory that grounds current understandings of digital media. Finally, understanding code as re-source links its effectiveness to history and context. If code is performative, it is because of the community (human and otherwise) that enables such utterances to be repeated and executed, that one joins through such citation.

This larger view of the performative has been developed by Judith Butler, who argues against that the felicity of a performative utterance does not depend on the sovereign subject who speaks it. Instead, she argues that what is crucial to a performative utterance's success or failure is its *iterability*, where iterability is "*the operation of that metalepsis by which the subject who 'cites' the performative is temporarily produced as the belated and fictive origin of the performative*."[62] In other words, when a speaker executes a performative utterance, she or he cites an utterance that makes "linguistic community with a history of speakers."[63] What is crucial here is: first, code that succeeds must be citations—and extremely exact citations at that. There is no room for syntax errors; second, that this iterability precedes the so-called subject (or machine) that is supposedly the source of the code; and third, and most importantly, an entire structure must be in place in order for a command to be executed. This structure is as institutional and political as it is machinic.

To make this point, I'll conclude by directly addressing an example that has "haunted" this essay: the current struggle between free software and open source software. At the material level, this disagreement makes no sense, for what is the difference between free and open source software, between linux and gnu linux? Nothing and everything—an imagined yet crucial difference. According to Richard Stallman, the difference lies in their values:

> The fundamental difference between the two movements is in their values, their ways of looking at the world. For the Open Source movement, the issue of whether software should be open source is a practical question, not an ethical one. As one person put it, "Open source is a development methodology; free software is a social movement." For the Open Source movement, non-free software is a suboptimal solution. For the Free Software movement, non-free software is a social problem and free software is the solution.[64]

62. Judith Butler, *Excitable Speech: A Politics of the Performative* (New York: Routledge, 1997), p. 51, emphasis in original.

63. Ibid., p. 52.

64. Richard Stallman, "Why 'Free Software' is better than 'Open Source.'" http://www.gnu.org/philosophy/free-software-for-freedom.html.

The fact that code automatically does what it says is hardly central. The difference between open source and free software lies in the network that is imagined when one codes, releases, and uses software—the type of community one joins and builds when one codes. The community being cited here, worked through, is one committed to this notion of freedom, to coding as a form of free speech. Open source and free software movements are aligned, however, in their validation of "open": freedom is open access.[65]

This emphasis on imagined networks I hope makes it clear that I'm not interested in simply exorcising the spectral or the visual, but am rather trying to understand how its spectrality lies elsewhere. Capturing ghosts often entails looking beyond what we "really" see to what we see without seeing, and arguably, digital media's biggest impact on our lives is not through its interface, but through its algorithmic procedures. Agre has emphasized its mode of capture rather than surveillance—that is, the ways in which computers capture our routine tasks as data to be analyzed and optimized, from shopping to driving a car.[66] Capture re-works the notion and importance of access: one does not need a "personal computer" in order to be captured—all one needs is a RFID tag or to work in a factory or to simply be in debt. Capture also emphasizes tactility and the mundane rather than the spectacular—but also the political and theoretical importance of imagining these invisible networks and technologies that envelope us in their dense thicket of passive and active signals, which can only be imagined.

This necessary imagining, this visually spectral other, underscores the fact that we do not experience technology directly, although to what extent human sensory experience or affect is the end all and be all of computation remains to be seen, for emphasizing human perception can be a way of clinging to a retrograde humanism. This necessary imagining also means that software can only be understood in media res—in the middle of things. "In media res" is a style of narrative that starts in the middle as the action unfolds. Rather than offering a smooth chronology, the past is introduced through flashbacks—interruptions of memory. To return to the parable of the six blind men relayed much earlier, this means that the position of the blind men who know without knowing is not one that can be superseded, but rather the exact position from which we can inter-

65. The question that remains, which is the subject of another paper, is: What does this meaning of open close down?

66. Philip E. Agre, "Surveillance and Capture: Two Models of Privacy," *Information Society* 10:2 (1994): 101–127.

vene and know. Software in media res also means that we can only begin with things—things that we grasp and touch without fully grasping, things that unfold in time—things that can only be rendered "sources" or objects (if they can) after the fact.