48

READING CULTURE THROUGH CODE

Mark C. Marino

Historically, computer source code has been treated like hieroglyphics read only by high priests or, in more mundane terms, like a highly specialized mechanism read only by technicians, even as software increasingly influences every aspect of our lives. However, in the last 10 years, reading code has become common among more and more groups, from a great variety of perspectives. In fact, 2012 was called the Year of Code, and sites such as Code Year offer online instruction in programming to the uninitiated. Courses that teach code have once again been added to elementary school curricula, to an extent not seen since the days of BASIC, Logo, and personal computing in the 1980s. In his 2016 State of the Union Address, U.S. President Barack Obama included computer science courses for all students as one of his chief priorities for the remaining days of his presidency. We are in the middle of a new wave of interest in programming, with a growing sense that programming, like mathematics and reading, is one of the core skills.

In the midst of the broader literacy movement (where "literacy" itself has been challenged as a term), I have been working with an interdisciplinary group of scholars to pursue the epistemology of code, asking whether it is more than merely a practical, utilitarian system of *encoding* and instead a form of *communication* with layers of meaning for its wide and varied audiences. We call these methods Critical Code Studies (CCS), which is the application of hermeneutics to interpret source code's extra-functional significance. Rather than treating code as the sole object and end point of this application, CCS sees it as an entryway into investigating the interactions between not only computers and humans but also code and many different kinds of systems, including software and hardware. Instead of rendering code an inevitable set of processes (such as arithmetic) or a logical extension of hardware, CCS frames code as a *cultural text*—not a fine art, but perhaps an artisanal craft. Donald Knuth (1992) coined the term "literate programming" to name the practice of writing code not merely for its function but also for its form, attending particularly to its legibility for human readers. Importantly, the act of programming does not involve the fulfillment of an obligation. Instead, it is the subjective and creative construction of an assemblage from a wide array of possibilities—an infinite array of them, in fact.

Programmers choose from various languages, programming styles, and architectures. From these selections, the programmer (alone or, more often, in collaboration with others) creates an assemblage of processes. For any given task, there are millions and millions of choices. Of course, the choice is not completely free. Each programming situation has its own constraints, including its own parameters and conventions. Hardware, time, money, social conditions,

cultural norms, and the priorities of a project may also limit or structure what programmers can do. But, like the chef working on a limited budget, constraints do not necessarily imply a lack of creativity or self-consciousness. Programming can be an imaginative act and intervention.

What Is a Critical Approach to Code?

How can CCS be distinguished from any discussion of code? If the process of contextualizing code involves an ethnographic or archaeological process, then CCS also presents an ontological challenge. One might think that an approach borne of hermeneutic or literary traditions would interpret code as a work of poetry, echoing the WordPress mantra, "code is poetry" (Bigelow 2011). Yet this suggestion makes some programmers cringe. Programming is not only prosaic; code can also seem virtually illegible, especially when written by someone else. Rather than interpreting code as a literary object or presupposing a high quality of expression, approaching it as a cultural text assumes that significance arises from its history, circulation, and reception. This approach draws heavily from cultural studies, as typified by a collaborative reading of the Sony Walkman (Du Gay et al. 2013). In this sense, a "text" is less an artwork and more an everyday object of inquiry—like a building, a nail, or even a network—with a meaningful relationship to culture. To treat code as a cultural text is to examine its accrual of significance through the ways it is regarded as well as the ways people respond to and interact with it.

In the realm of computer science and programming classes, code is already more than a utilitarian medium, for it has subjective, aesthetic traits. Elegance, for example, is a subjective quality of code that is routinely praised. While I am not particularly interested in debating what makes code elegant, the fact that programmers develop a *sense of elegance* and pursue it suggests a main attribute of programming: it is not a purely functional activity, and the shape of the code is not inevitable. To make code is to engage in discourse while also recognizing that code has to function, even if it's irreducible to that functionality. Programmers speak code and code speaks, as Geoff Cox and Alex McLean (2013) put it. Code proves itself all the more discursive because it is made of signs—material signs in artificial languages—and, like any realm of discourse, this one develops rhetoric, creativity, and meaning.

In pursuing this cultural approach, CCS builds upon the practices of material culture studies, media archaeology, and science and technology studies (Spiegel-Rösing & de Solla Price 1977) by emphasizing critical intervention. Though such approaches involve a wide array of practices from scholars whose foci hardly follow a single rule, the comparison is instructive in identifying high-level differences in emphasis. For, in keeping with the historical inquiry of these three approaches, CCS seeks to recover the conditions under which code has been written, not primarily for archival purposes, but in service of inquiry into the more complex aspects of history: the assumptions behind code's creation; the negotiations of forces, social and physical, that enabled or inhibited its production; and the nature of its reception and iteration, to name a few. CCS pursues the "throughlines" of technological development, where even an artifact as young as code participates in histories that long predate it. A goal of CCS is thus *not merely to document but to question*—to ask what is this object, where did it come from, how did people develop it, how did people use it, and how was it received. While CCS demands a kind of ethnographic and, following the methods of Matthew Kirschenbaum (2008), forensic research, the term "critical" implies a challenge to conventional narratives of technological development. Rather than tracing the history of media forms for the sake of nostalgia, corporate history, or some progress narrative, CCS investigates and critiques the forces that shape technology as well as the investments and motivations of those,

at times, agonistic forces. Such investigations can be equally applied to one line of code and to thousands of lines (see Montfort et al. 2013).

Still, even if some programs are as short as some poems, code cannot be completely understood by reading its text alone. Code is not simply what is seen (symbol or signifier); it shapes what is seen. Early in the development of CCS, Kirschenbaum (2011) pointed out that code cannot be read by itself, separate from its platforms and the other software with which it interacts. Without these platforms and dependencies, it will not execute. Instead, we read code in situ, in the context of its platform and interoperating software, as well as the culture of its development and circulation.

After situating code within its cultural and technological context, we determine what code does and how. Perhaps this is the biggest difference between reading code and reading other text-based semiotic objects. As part of a digital system, code has specific and predictable effects on the interoperating software as well as the hardware on which it runs. Unlike a poem or even a law, which exist with typographical errors, a line of code rarely functions with a mistake. The systems that process code are largely unforgiving of error. Code may be, as Alexander Galloway posits, "*the only language that is executable*" (2004: 165–66); it is a language that, when processed (whether compiled or executed), causes certain necessary, deterministic effects. Machines process it in a verifiable and replicable manner, with input becoming output. In CCS, the critic must know which effects code causes in the system, while recognizing that every known effect stands in for and masks a host of other effects. Here, Wendy Chun's warning against "sourcery" (2011), or fetishizing source code as the ultimate underlying truth, is both a necessary caution and a reminder not to see code as a sole means or a definitive end. Nonetheless, to use and read code competently and consciously, one must understand its parameters, conventions, and effects. As a result, CCS involves some fluency in programming as well as working with those who have an intimate knowledge of language and programming paradigms. It is a moment in digital humanities that calls for collaboration, through which interpretation takes advantage of the same diversified skillsets that go into building software.

Until this point, my description of CCS has largely paralleled strains of media archaeology and science and technology studies. However, CCS also follows in the footsteps of critical race studies and critical legal studies in its calls for reflexive interpretation. That is, the "critical" dimension of CCS refers to not only a critique of conventional narratives (as mentioned earlier), but also the use of critical theory—a term applying loosely to philosophical hermeneutics (or "ways of interpreting") largely arising from Continental philosophy—as well as the critical approach of the examiner (or "theories of knowing"), including issues of ethics, social relations, ideology, or other aspects of epistemology and ontology. This approach presupposes that all is not apparent on the surface of code. Again, code means more than it shows as a symbol or signifier, and the conventions that frame how we interpret it are also constructed.

This challenge to conventional wisdom that code is what it does (i.e., its function) arises from a deconstructive approach. Associated with Jacques Derrida (1974), deconstruction is a practice of identifying fissures or gaps between the signifier and signified and then re-inscribing those gaps as resources fundamental to understanding the text at hand. While programmers must ask what any particular line of code does and how might it help them achieve a goal, deconstruction asks how the essence or source of code actually depends on its appearance or effects. Deconstruction does not take conventions or origins as gospel. It questions the underlying assumptions of those conventions and origins. Out of this deconstructive impulse

comes Friedrich Kittler's claim that "there is no software" (1995), Wendy Chun's note that the source code is not exactly executable code (2011), and Tara McPherson's analysis of Unix and the Civil Rights movement (2012). Even if deconstruction has fallen out of favor, it is hard to imagine most contemporary forms of critical interpretation without its contributions to reading as a practice predicated on productive skepticism. The examples I have offered via Kittler, Chun, and McPherson show how critics of media and technology problematize simple interpretations by destabilizing systems of meaning, including the empirical "facts" or "natures" of computational systems.

While these gestures challenge the authority or essence of code, they do not detract from its ability to convey meaning. Ultimately, CCS must seek meaning in code. But where? In the comments? In the variable names? In how it executes processes? In the output or results? Obviously, there is no one answer. Meaning emerges from all of these aspects, yet CCS does not define code as a mere cipher, whose meaning need merely be revealed. Reading code, like reading any cultural text, is a creative and relational process, in which meaning arises from questions asked.

Case Studies in Reading Code

To illustrate some CCS methodologies, I will present a few examples drawn from my own work as well as from the CCS Working Groups, which are biennial online conferences that, since 2010, have gathered scholars from around the world to study culture through code. These working groups have helped grow CCS into a vibrant subfield such that code analysis is now a chief methodology of digital humanities. The case studies I have included here address code through examples that are relatively self-contained and easy to explain in a concise manner. As experimental works, they also perform a certain amount of commentary on code. Analyzing them thus offers a twofold critique, both the critique the works perform and the one emerging from my interpretations of them against various cultural contexts.

One week in the 2014 CCS Working Group focused on reading code through post-colonialism to critique and explore issues of power, subjugation, and identity formation after the period of large-scale colonial rule. The leaders of the discussion, Roopika Risam, Adaline Koh, and Amit Ray, examined the predominance of English in programming languages, by examining قلب or Alb ("heart"), a work presenting itself as an Arabic programming language (reposted 2014). Created by computer scientist and artist, Ramsey Nasser, in 2013, قلب is "a conceptual art piece exploring the difficulty of using any language other than English for practical software" (Nasser in McBride 2013). According to Nasser, the language is "LISP-based and similar to a variant called Scheme." Here is some sample code. Note the persistence of Roman characters and English words, such as "class," even in this snippet.

```
(٠،١،٣ مترجم - لغة برمجة :"قـلـب" قول")
(٢٠١٢ رمـــــزي نـــاصـــــر" قول")
(قول "")
(قول "<a class='execute' href='#أمثلة'>أمثلة</a> - <a class='execute'
href='#النجـدة'>النجدة</a> - <a class='execute' href='#ما هـذا؟'>ما-هذا؟</
a>")
(قول "")
```

```
(لامدا) حدد أمثلة (لامدا () (
  (قول "سهل: <a class='load' href='#amthila/mrhba'>مرحبا يا عالم</a>")
  (قول "متوسط: <a class='load' href='#amthila/fybunatshi'>عدد فيبوناتشي</
a>")
  (قول "متقدم: <a class='load' href='#amthila/konway'>لعبة الحياة لكونواي</
a>")
  (عدم)
)))

(("حدد النجدة (لامدا () (قول "هذا قلب!"))
(لامدا) حدد ما-هذا؟ (لامدا ()
  ("قول "قلب هي لغة برمجة هدفها إستكشاف)
  ("قول "العلاقة بين الثقافة البشرية)
  ("قول "والعلوم الحاسوبية، خصوصًا علاقة)
  ("قول "الأبجدية اللاتينية بلغات البرمجة)
  ("قول "المختلفة. للمزيد من المعلومات)
```

In many ways, Nasser's project already enacts a form of CCS by identifying the cultural dimension of programming, which is often considered purely logical or universal. However, the stakes are more apparent in an economic landscape, where many jobs in the technology sector depend on mastery of English-based programming languages (Vee 2013). Furthermore, قلب demonstrates programming's expressive potential, or how code is a creative medium for cultural intervention.

To discuss this code, the CCS hosts drew upon postcolonial theories of language, primarily Edgar Schneider's theorization of global Englishes in *Postcolonial English* (2007). In his analysis, Schneider maps the progress of English into a worldwide language through British colonization. Risam, Koh, and Ray make the argument that English has followed a similar course through programming languages, establishing its predominance specifically in American English. For instance, consider English words used in high-level programming languages (e.g., PRINT, if/then/else) as well as the left-to-right flow of code. However, programming languages do not operate through direct political colonization. Instead, their pervasive use in software makes them the lingua franca of the computational world. Risam, Koh, and Ray draw an argument from another trio, Bill Ashcroft, Gareth Griffiths, and Helen Tiffin, who, in *The Empire Writes Back* (1989), describe "a process through which writers disrupt the dominance of colonial English through forms of lexical resistance. Such writers, they suggest, take up the act of abrogation, refusing the dominant aesthetics and categories of imperial culture . . ." (Risam et al. 2013). By situating Nasser's code as an analogous act of abrogation, of disruption through code, these scholars gesture toward other potential acts of resistance to the colonizing force of English in programming languages.

When posing such questions of code and software, postcolonial criticism does something radical or perhaps unthinkable in work that adheres strictly to a particular discipline. In many ways, Nasser's code may be compared to the disruptive force of Rushdie's *Midnight's Children* (1981), Rhys's *Wide Sargasso Sea* (1966), or Achebe's *Things Fall Apart* (1958), as it resists popular colonial narratives. Nasser's project, together with its explication by Risam, Koh, and Ray, demonstrate how CCS extends discussions of programming by adding critical lenses that speak to other realms of communication.

If قلب and Risam, Koh, and Ray's interpretation disrupt programming by repositioning it in postcolonialism, then the next example resituates global positioning as part of geopolitical theater.

Transborder Immigrant Tool

In 2010, the Electronic Disturbance Theater (EDT) released the code for their provocative Transborder Immigrant Tool (TBT), a mobile phone application designed to help travelers who had crossed the Mexico-U.S. border survive their final mile's journey through the desert by offering them poetry and directions to water stations. Written mostly by Amy Sara Carroll, the poems take the form of prose modules, which offer guidance on how to survive in the desert, woven out of varied sources of knowledge, cultural allusions, and compelling imagery.

The code for the project, which was released before the project had been fully implemented—or, as I have argued elsewhere (Marino 2013), was released as a part of the piece's implementation—was written in Java, specifically to work on the J2ME platform of the Motorola i455 phone, chosen because it is inexpensive and relatively robust. In addition to the J2ME libraries, the code takes advantage of GPX, a platform Brett Stalbaum worked on to facilitate walking adventures. Importantly, this app was developed for a platform that predated and actually anticipated technologies we now take for granted. Stalbaum wrote much of the code, collaborating with Jason Najarro, an undergraduate at University of California, San Diego (UCSD) (Marino 2011a). Working on this code offered Najarro a chance to develop his skills for his résumé. However, that plan went astray when a *Vice* magazine piece about the TBT was picked up by conservative bloggers and then by conservative news outlets, specifically Glenn Beck and Fox News. The outrage did not end there, as Todd Hunter and other U.S. politicians used the story to assail the public institution (UCSD) that funded the work (Hunter 2010). The controversy inevitably led to an audit of the TBT project and a fuller inquiry into the artistic activities of EDT member, Ricardo Dominguez (Su 2010).

This disruption was precisely the drama the EDT (consisting of micha cárdenas, Carroll, Dominguez, Elle Mehrmand, and Stalbaum) intended. In a personal interview (2011b), Dominguez explained to me that the TBT was developed as a Mayan technology in the tradition of the Zapatistas, conceived to confront and then confound authorities and other audiences. In that context, even the outrage of the politicians and pundits became part of the performance (Su 2010). However, the publicity also incited hate toward the artists, specifically physical threats directed toward the members and their families. Needless to say, the group had touched a nerve.

TBT's code offers conceptual frameworks that are not apparent in its audio or visual output. Most interesting are the metaphors used to name the functions related to finding water. Rather than using technical terms (e.g., "GPS" or "hotspot") or abstract names (e.g., "location" or "destination"), the team drew upon metaphors of "water witching." Water witching refers to finding water by means of a divining rod or dowsing compass in the form of a twig or stick. In contrast with technologies such as satellite positioning and cellular telephony, it uses parts of nature (e.g., broken branches) to find other parts of nature (e.g., the water which helped it grow). Consider the Java code activated when a water cache is found:

```
public void witchingEvent(TBCoordinates mc) {
    aheadCoords = mc;
    if (display.getCurrent().equals(tbDowsingCompass)) {
        waypointAheadAlert.setString(tbDowsingCompass.getInfo(mc));
```

```
waypointAheadAlert.setImage(aheadCoords.getIcon());
double distance = tbDowsingCompass.distanceTo(mc);
```

Although a full explanation of Java and object-oriented programming is beyond the scope of this chapter, "public" means this method can be called or accessed by any other class of objects. "Void" means this method does not return any values: it has no output to whatever code called it. This method takes two arguments: "TBCoordinates" and "mc." "TBCoordinates" represents where the traveler is; "mc" represents the destination of the traveler. A variable, "aheadCoords," is set to the target. Then, if the display is up to date, the waypoint is set and an alert is also set in the form of an icon. While the alerts are palpable (i.e., electronic notifications), the concept in the name of the function, "witchingEvent," is metaphysical.

More than just an arbitrary string, the method offers a new defining metaphor or conceptual framework (i.e., water witching) for how the app works. When a user arrives at the navigation point, this J2ME code is executed for an alert:

```
public void arrivedAtTarget(int distance) {
navigating = false;
// stop the compass from navigating
tbDowsingCompass.stopNavigation();
display.setCurrent(arrivedAlert);
display.vibrate(1000);
playAudioFile("arriving.wav", false);
}
```

Like the divining rod, the app makes the phone respond viscerally to proximity to water. It vibrates and plays an alert ("arriving.wav"), the physical signal necessary because the user may be undergoing heat exhaustion or heat stroke.

By turning a mobile phone into a dividing rod, the code also turns circuitry into a stick. But the stick does not just find water. For the person reading the code, it rearranges her relationship to the political drama of the Mexico-U.S. border—changing the tale from one of transgressions to one of survival. It rewrites the geopolitical narrative into a story that facilitates human sympathy and empathy. In fact, the code presents various scenarios, including one in which the traveler successfully reaches water:

```
waypointAheadAlert = new Alert (translation.translate("Site Ahead!")
arrivedAlert= new Alert(translation.translate("Arrived at Site")
```

A very different scenario is depicted in the code when the traveler stops moving:

```
If (isMoving) {// updated to moving
nearbyWPList =
tbDowsingCompass.getNearbyWaypoints (SEARCH_DISTANCE);//so update
nearby point
}else { //updated nt moving
display.vibrate(200).
If (moveWarningEnervator % 5 == 0) {//only play this file ~ every 5th time
playAudioFile("move.wav", false);//the "move for compass message can be too
frequent
```

In the first scenario, the alerts are sounded, the phone vibrates, and messages are sent, indicating that the waypoint has been found. In the second case, "movWarningEnervator" is triggered to keep the traveler going—to survive. It is hard to miss the pathos in this moment of code, demonstrating the emotional impact of its framing of a desperate moment in the life of its user.

Let us look at one more example to show how CCS can be used in conjunction with a variety of other approaches.

Tachistoscope

For 6 years, I have collaborated with Jessica Pressman and Jeremy Douglass to examine William Poundstone's work of electronic literature, *Project for Tachistoscope {Bottomless Pit}* (2005), which was composed in Adobe Flash. During the process of analyzing Poundstone's work, each of us took to our preferred methods of approaching a digital object, sharing our findings with the others to build collaborative readings. Douglass used various computational analysis methods, particularly visual analytics. Pressman used textual analysis and media archaeology, hunting through patents and other scholarship on the tachistoscope. To these, I added my analysis of the code. What we found in the code seriously affected the way we interpreted *Project*.

Project presents the story of a mysterious bottomless pit by displaying one word at a time in rapid succession on a screen. It is a busy work that challenges the reader to pay attention. In this way, it is similar to Young Hae Chang Heavy Industry's works, such as *Dakota* (2002; also see Pressman 2014). While flashing these words, *Project*'s interface also overlays a set of visual effects. At times, one can even perceive, though not consciously read, a set of words flashing in between the story words. The various introductory texts, accessible before the reader presses the "START" button, situate these as subliminal words, designed to prime the reader. Pressman pointed us toward the paratexts, or entry screens, that referred to this method of priming. Douglass's visual analysis of screenshots identified these subliminal words that seemed randomly drawn from a set. He asked me what I could find in the code about these subliminal words.

As it turns out, the code expresses a very different metaphor: spam. As in several of the previous cases, the metaphor resides in the most arbitrary aspect of the code: the variable and function names. In the ActionScript code of Poundstone's Flash file, the story's text is loaded from a TXT file into two variables. The story text is loaded in "storyvar," and the subliminal words are loaded into "spamvar" (not, as one might expect, "subliminalvar"). These rapidly interspersed words are not mere add-ons or incidental distractions but are instead central to the code's function: the call to display the spam initiates the story. In fact, the very first word displayed is not a story word but a spam word: "elongate."

When I revealed these findings to my collaborators, we found that it changed the way we interpreted the story, especially when processing the subliminal words. If subliminal messages claim to manipulate our subconscious desires, then spam are messages we consciously wish to block. If subliminal messages attempt to slip past our conscious blocks, then spam attempt to slip past our email filters. This new classification further helped us understand why the words have so little to do with the story or even each other, for that matter. Douglass noted that the spamvar words resemble the lists of words used at the end of spam emails to jam software that filters out spam. In a presentation in 2015, Poundstone confirmed these words were drawn from such an email. In light of the new conceptual framework of spam, the words changed their status in our reading—from hidden, barely perceived suggestions to

egregious unwanted junk mail; from priming words to unsolicited messages. I should also mention that the hunt for the code and source files yielded an easy-to-read list of the subliminal and story words, a discovery which freed us from *Project*'s one-word-at-a-time animation.

Examining the code did not just affect the way we regarded the text. It also changed how we considered the interface effects that play throughout *Project for Tachistoscope*. While reading the code, I thought the distraction effects—or the flashing spam words—pulled the reader away from the primary text, or the story words. By extension, Poundstone's piece seemed to perform the war on attention that is so prevalent in digital environs (Hayles 2007; Davidson 2011). For example, one set of effects flashes the spam words in white. A set of white crosses, known in the code as "fixationCrosses," prime the reader to be attentive to these white words. (The term, "fixation crosses," references the small plus signs used as focal points in instruments that measure vision.) In *Project*, we initially thought the spam words *only* appear when they flash in white. However, we later realized they are in fact continuously flashing. Here is the ActionScript that renders the results:

```
if (index>.50) {
level = 3;
_level0.centerPoint.level2Effect_mc.unloadMovie();
var hue = Math.floor(Math.random()*(noOfColors+1));
myColor.setRGB(palette[hue]);
mySubliminalColor.setRGB(0xFFFFFF);
fixation_mc.attachMovie("fixationCrosses", "myFixationCrosses", 300);
}
```

Consequently, even the term "fixationCrosses" is a bit deceptive. It obscures how this piece directs and misdirects attention. It causes the eye to fixate on the white (FFFFFF) and notice the white words that follow. Yet, at the same time, it obscures the black words. Once again, Poundstone's piece is not only directing attention but also distracting the reader from noticing the subliminal words flashing by.

"fixationCrosses" helped us understand how Poundstone is playing with the language and conventions of tachistoscopes to challenge our relationships with the "flickering signifiers" of digital culture (Hayles 2008). By repurposing the tools of focus—the fixation crosses—Poundstone's version of a tachistoscope demonstrates how signifiers relayed for swift viewing entice a rich and methodical slow reading: the way those icons pull our eyes may be priming us to recognize one set of signs while obfuscating another. Only by looking at *Project*'s source files and code could we see how he confounded our perception. And yet, identifying this trick of the eye prompted us to examine this work more closely, using additional reading methods and machines to read the source code.

Note how this reading engages not just the natural language used to name objects and variables but also the realizations that can only be reached by examining code. Guided by the questions and observations of the other methodologies (i.e., visual analytics, textual analysis, and media archaeology), the examination of code yielded an insight into the design strategies of the piece. The code proves *Project* to be a distraction machine, but also a meditation on the nature of focus in a digital space. Similar to the analysis of the TBT, the code offers the conceptual metaphors used by designers, adding a layer of meaning to our interpretation of the piece. In this way, the study of code does not merely resume or promote the hunt for artistic intention. However, inasmuch as code is a material and mechanized manifestation of thought, it offers conceptual frameworks, organizational hierarchies, and ways of rendering

that do more than merely complement the output of software—they define it. To study the code is to engage with these ideas as they express themselves through particular social and material conditions.

★ ★ ★

These examples demonstrate how CCS can be used in tandem with other approaches to develop multifaceted interpretations of texts and culture, where code is treated as a cultural text. They are not offered to delimit CCS—to draw a border around it—but instead to open it up and incite further exploration as well as further collaboration between those who build and those who interpret—*between the methods of making and partaking*—in the hope that we can realize how our technologies and our techniques for understanding them are mutually implicated and inextricably entwined.

Further Reading

Berry, D. M. (2011) *The Philosophy of Software: Code and Mediation in the Digital Age*, New York, NY: Palgrave Macmillan.

Brown, Jr., J. J. (2015) *Ethical Programs: Hospitality and the Rhetorics of Software*, Ann Arbor, MI: University of Michigan Press.

Chun, W. H. K. (2011) *Programmed Visions: Software and Memory*, Cambridge, MA: MIT Press.

Coleman, E. G. (2012) *Coding Freedom: The Ethics and Aesthetics of Hacking*, Princeton, NJ: Princeton University Press.

Cox, G. and A. McLean (2013) *Speaking Code: Coding as Aesthetic and Political Expression*, Cambridge, MA: MIT Press.

Jerz, D. G. (2007) "Somewhere Nearby Is Colossal Cave: Examining Will Crowther's Original Adventure in Code and in Kentucky," *Digital Humanities Quarterly* 1(2), retrieved from www.digitalhumanities.org/dhq/vol/1/2/000009/000009.html.

Vee, A. (2017) *Coding Literacy: How Computer Programming Is Changing Writing*, Cambridge, MA: The MIT Press.

References

Achebe, C. (1958) *Things Fall Apart*, London, UK: Heinemann.

Bigelow, S. (2011) "Code Is Poetry, CSS Is Art," *WordPress.com New*, retrieved from en.blog.wordpress.com/2011/11/24/code-is-poetry-css-is-art.

Chun, W. H. K. (2011) *Programmed Visions: Software and Memory*, Cambridge, MA: MIT Press.

Cox, G. and A. McLean. (2013) *Speaking Code: Coding as Aesthetic and Political Expression*, Cambridge, MA: MIT Press.

Davidson, C. N. (2011) *Now You See It: How Technology and Brain Science Will Transform Schools and Business for the 21st Century*, reprint edn, New York, NY: Penguin Books.

Derrida, J. (1974) *Of Grammatology*, G. C. Spivak (trans.), Baltimore, MD: Johns Hopkins University Press.

Du Gay, P., H. McKay, K. Negus, L. Janes, and S. Hall (2013) *Doing Cultural Studies: The Story of the Sony Walkman*, 2nd edn, London: Sage Publications.

Galloway, A. R. (2004) *Protocol: How Control Exists after Decentralization*, Cambridge, MA: MIT Press.

Hayles, N. K. (2007) "Hyper and Deep Attention: The Generational Divide in Cognitive Modes," *Profession* 13, 187–99.

Hayles, N. K. (2008) *How We Became Posthuman: Virtual Bodies in Cybernetics, Literature, and Informatics*, Chicago, IL: University of Chicago Press.

Hunter, D. (2010) "Taxpayers Should Be Outraged at this Use of Funds," *San Diego Union-Tribune*, retrieved from www.sandiegouniontribune.com/news/2010/mar/07/taxpayers-should-be-outraged-use-funds.

Kirschenbaum, M. (2008) *Mechanisms: New Media and the Forensic Imagination*, Cambridge, MA: MIT Press.

Kirschenbaum, M. (2011) "<!—opening thoughts—>," Critical Code Studies, *HASTAC*, retrieved from www.hastac.org/comment/3711#comment-3711.

Kittler, F. (1995) "There Is No Software," *CTheory*, retrieved from www.ctheory.net/articles.aspx?id=74.
Knuth, D. E. (1992) "Literate Programming," *CSLI Lecture Notes, No. 27*, Stanford, CA: Center for the Study of Language and Information.
Marino, M. C. (2006) "Critical Code Studies," *electronic book review*, electropoetics, retrieved from www.electronicbookreview.com/thread/electropoetics/codology.
Marino, M. C. (2011a) Interview with Brett Stalbaum.
Marino, M. C. (2011b) Interview with Ricardo Dominguez.
Marino, M. C. (2013) "Code as Ritualized Poetry: The Tactics of the Transborder Immigrant Tool," *Digital Humanities Quarterly* 7(1), retrieved from www.digitalhumanities.org/dhq/vol/7/1/000157/000157.html.
McBride, S. (2013) "Arabic Code Language Won't Make Waves: Creator" ITP.net, retrieved from itp.net/592111-arabic-code-language-wont-make-waves-creator.
McPherson, T. (2012) "U.S. Operating Systems at Mid-Century: The Intertwining of Race and UNIX," in L. Nakamura and P. A. Chow-White (eds.) *Race after the Internet*, New York, NY: Routledge, 21–37.
Montfort, N., P. Baudoin, J. Bell, I. Bogost, J. Douglass, M. C. Marino, M. Mateas, C. Reas, M. Sample, and N. Vawter (2013) *10 PRINT CHR$(205.5+RND(1));_: GOTO 10*, Cambridge, MA: MIT Press.
Nasser, R. (2013) قلب, retrieved from nas.sr/%D9%82%D9%84%D8%A8/.
Obama, B. (2016) "Remarks of President Barack Obama—State of the Union Address As Delivered," Whitehouse.gov.
Poundstone, W. (2005) *Project for Tachistoscope {Bottomless Pit}*, www.williampoundstone.net.
Pressman, J. (2014) *Digital Modernism: Making It New in New Media*, New York, NY: Oxford University Press.
Pressman, J., M. C. Marino, and J. Douglass (2015) *Reading Project: A Collaborative Analysis of William Poundstone's Project for Tachistoscope {Bottomless Pit}*, Iowa City, IA: University of Iowa Press.
Risam, R., A. Koh, and A. Ray (2013) "PoCo Critcode: Coding in Global Englishes," Critical Code Studies Working Group, retrieved from roopikarisam.com/uncategorized/coding-in-global-englishes.
Rhys, J. (1966) *Wide Sargasso Sea*, London, UK: André Deutsch.
Rushdie, S. (1981) *Midnight's Children*, New York, NY: Random House.
Schneider, E. W. (2007) *Postcolonial English: Varieties around the World* Cambridge, UK: Cambridge University Press.
Spiegel-Rösing, I. and D. J. de Solla Price (1977) *Science, Technology, and Society: A Cross-Disciplinary Approach*, International Council for Science Policy Studies, London, UK: SAGE Publications.
Su, E. Y. (2010) "'Activist' UCSD Professor Facing Unusual Scrutiny," *San Diego Union Tribune*, retrieved from www.sandiegouniontribune.com/news/2010/apr/06/activist-ucsd-professor-facing-unusual-scrutiny.
Vee, A. (2013) "Understanding Computer Programming as a Literacy," *Literacy in Composition Studies* 1(2), 42–64.
Young-Hae Chang Heavy Industries (2005) *Dakota*, www.yhchang.com.