Transcoding Sexuality: Computational Performativity and Queer Code Practices

Gerald Stephen Jackson

QED: A Journal in GLBTQ Worldmaking, Volume 4, Number 2, Summer 2017, pp. 1-25 (Article)

Published by Michigan State University Press

Transcoding Sexuality: Computational Performativity and Queer Code Practices

Gerald Stephen Jackson

ABSTRACT

Scholarship in the history of computer programming has demonstrated how the contributions of engineers have been erased due to gendered assumptions of labor and science. I argue that such erasure parallels a gendered epistemology of mastery embedded in computer programming itself. Towards this, I extrapolate and refine the concept of "computational performativity," drawing from media theorists Katherine Hayles and Wendy Chun, to put forth a gender performativity of code. Looking at two queer code art objects—Zach Blas's *transCoder* and Julie Levin Russo's *Slash Goggles*—as well as production code in the C programming language, I argue for a critical move away from representation as the seat of meaning in code, and towards a performative understanding of gendered code through "contexts of complexity." Focusing on complexity and interrelationships allows scholars to read code as gendered, and to furthermore leverage such reading to propose and enact queer "failures" in code as rhetorical critiques of software.

))) Introduction

Digital media is not gender neutral. Although theorists in feminist technology and queer studies have illuminated the social construction of technological objects, many arenas of software development still posit a gender-blind meritocracy of progress and achievement. The historical and epistemological assumptions

of the field contradict such utopian visions. As Wendy Chun writes, the original computers were in fact women, literally computing problems given to them by (typically male) mathematicians and scientists, plugging away on 1940s state-of-the-art military-grade hardware.[1] This man-woman-machine relationship is the precursor to the development of software, and as women were marginalized from the burgeoning field of "programming," so too was digital media bracketed as prior to or outside gender, effectively moving it from the clerical realm of hardware manipulation into the creative and intellectual realm of computer science.[2] Because women, in many ways, were software before there was software, scholars can see digital media as an inherently gendered entity and develop methods to study technology as a productive and performative aspect of cultural epistemologies. Not only does code represent the instructions that eventually come together to form software, but it also expresses a set of knowledge-producing practices that join programmers and users through a plethora of technological enterprises.

In this article, I argue that one such epistemology, that of "mastery," exemplifies the performative and gendered nature of software and code practices. I situate programming as a profession of software production, the result of performative norms of humanistic mastery of discourse and technology. Towards this, I extrapolate and refine the concept of "computational performativity," drawing from media theorists Katherine Hayles and Wendy Chun, to put forth that computational performativity *is* gender performativity. Rather than see code as a representation of meaning through the interpretation of texts or processes, I utilize several code artifacts to show that software is more accurately examined as an ecology of development practices that structure how certain norms circulate through software. These practices can be traced through the distinction between code as a text (source code, algorithms) and code as software—as made from the relationships of multiple, often hidden processes. Inspired by Judith Butler's assertion that "the linguistic domain over which the subject has no control becomes the condition of possibility for whatever domain of control is exercised by the speaking subject," I trace how computational performativity functions as a technological production of structural masculinity through software.[3] Moving away from representation as the seat of meaning in code, I see the reading of code as a formal tool to discuss how gender works through what I call "contexts of complexity." Complexity, in software engineering, is a constant problem and mechanism that requires that programmers strategically hide complexity to produce further, complex software. To read a piece of code as hidden and hiding, always embedded in contexts beyond its immediate application, is essential to understanding how assumptions of discourse propagate through that hiding and production.

To illustrate my vision of computational performativity, I examine two related digital artifacts: Julie Levin Russo's gender-challenging *Battlestar Galactica* fan fiction *Slash Goggles* and Zach Blas's digital queer art project *transCoder*. A fictional "pseudocode," *Slash Goggles* is an algorithm meant to bring together elements of computation, fiction, and gender critique. *Slash Goggles* mobilizes computer programming structures and technical communication to present a theoretical piece of software that operationalizes sexual identification through characters in the show. The programming syntax used as the fictional "language" of *Slash Goggles* is obtained from *transCoder,* a digital art project that synthesizes software development techniques and queer gender theory from thinkers such as Butler, Luce Irigaray, Michel Foucault, and Donna Haraway. Using *transCoder* as a template for *Slash Goggles*, Russo develops her project within a set of computational norms and fictional contexts of complexity that perform traditional software engineering practices. As a consequence, *Slash Goggles* implicates her critique and her fan fiction within a larger critique of digital circulation and sexual identity.

The move from a template library of code in *transCoder* to a realized procedural critique in *Slash Goggles* demonstrates the performative move of software in iterating social and computational norms, and the potential for gender norms to likewise iterate through digital media. In making this argument, I proceed in three parts. First, I define the concept of computational performativity through *transCoder* and *Slash Goggles*. I show that the former structures the latter through what Hayles describes as the work of "revealing and concealing" in software, or the ways in which the "'brute' lower levels" of computation are concealed to engender higher-level order through a mechanism of "abstraction."[4] Second, I describe how the relationship between abstraction and complexity in code is a performative aspect of the circulation of a historically masculinized epistemology of mastery, in which the subject position articulated by programming as a practice rests on assumptions of expertise, individuality, and command. Third, I demonstrate how their queer critiques of code open the way for a rhetorical approach of computational performativity by way of the discursive potential of what J. Jack Halberstam defines as "queer failure" in code. Abstraction as performativity and queer failure allow us to recognize code as a product and production of a masculine epistemology, to critique that epistemology, and recognize the productive aspects of that epistemology.

))) Computational Performativity and Contexts of Complexity

Describing code as "performative" inevitably asks that we think of code as a kind of language, which for some is a contentious analogy to make. Because code is not "natural" (i.e., human) language, some overwhelmingly see that it is immediately constrained by its determined, machinic limitations as compared to natural language. Hayles argues that we cannot describe code in the same way as a natural language because code has a more rigorously defined referent in the form of hardware and execution structures, of esoteric commands, binary digits, code libraries, and electrical pulses—referents that have no antecedents in natural language.[5] For Hayles, the "effects" produced by computer code invoke, through the execution of code as part of software, an unintelligible interlocutor in the form of the machine. Perhaps because of this, scholars often productively turn to the ways in which code acts as a form of writing grounded in recognizable writing practices. Robert Cummings, for example, argues that the purposing of code as a text allows us to better understand how we write for human audiences, that there is a communicative relationship between machine structures and human writing. The machinic elements of this equation (the programming language, the software) are—even though they structure the entire activity—peripheral to the actual activity of writing the code (which, ideally, adheres to relatively understandable paradigms of rhetoric, composition, and pedagogy).[6] Likewise, code studies scholar Mark Marino suggests that we can "read" code (or pseudocode) like a poem, "a text, a sign system with its own rhetoric, as verbal communication that possesses significance in excess of its functional" utility.[7] This can be a strict representational look at the code (what does it say), or, as Ian Bogost or Kevin Brock suggest, we can look at the rhetoric of an algorithm through the procedure outlined by the code or expressed by the software (what does it do, or what does it ask the user to do).[8] Textual and procedural analyses therefore situate procedure, algorithm, or code qua text as writing, distinct from other labor practices in software engineering involved with creation of hardware or software. Writing would no doubt be part of such professional practices, and perhaps even more noticeably so when focusing on individuals writing code, but there is always some line of intelligibility that comes with describing software or computation itself as rhetorical.

Coding as a practice does not necessarily map onto a human-centric practice of writing, primarily because the responsibility of its interpretation also includes a machine: the computer. We cannot reduce code to textual interpretation, which in and of itself is not a world-shaking proposition considering that many

might argue we cannot reduce *text* to some paradigm of textual interpretation. However, it does suggest that, like language, we can also consider code as something beyond its representational force or content, whether considered through its text or its procedure. That is, how can we think of what code "does" as an object with a history, emerging from cultural contexts that rely just as much on software engineering techniques as they do cultural interpretation? I suggest that code is performative in the sense that I read code less for how it affects an audience, and more about what Butler describes as "the reiterative and citational practices by which discourse produces the effect that it names."[9] Following this, reading a piece of code as a text implicates more than the rhetorical force of the code itself, but the technocultural networks of code libraries and engineering practices that produces the potential meanings already situated in cultural categories of race, class, gender, and nationality.

Software is not "an algorithm" or code, but a network of code libraries producing contexts of complexity for execution that cannot be limited by an "audience–interlocutor" model, nor reduced to a mediation within such a model. In the examples of *Slash Goggles* and *transCoder*, computational performativity is demonstrated through the artwork of queer digital activists' representations of code and software, and an investigation shows how computational performativity can produce gendered effects. Because Hayles qualifies the notion that code is a language by implicating the computer itself as an audience, she qualifies code as a mediation of intention and logic between humans and machines. She accordingly resists a Derridean notion of language's infinite citationality and iterability—because code is tied to a knowable context (that of execution, of the machine), at some point "signifiers must point to signifieds," and commands must work within a tight constraint of "correctness" to execute.[10] Derrida's description of language as always a citation of previous utterances—utterances that iterate outside of the presences of their author—ends logically at the artifice of hardware. Therefore, the *writing* of code is not directly connected to the execution of that code through that act of writing: there are intermediary steps during the computational process that programmers and writers do not engage, which seemingly ends the endless play of signification present in natural language.

Within this discussion of code and execution, note that when Cummings, Marino, and Hayles refer to "code," they mean "source" code. Source code is the code that programmers write, modify, and compile as part of the software development process. This differs from machine code, which is expressed in binary or hexadecimal numerical form and corresponds directly with the processes occurring in computer processing and memory. Nontechnical audiences

are familiar with source code through representations of programming in popular media, with focused hackers, spies, or scientists typing away while lines of mathematical-looking statements scroll down a screen. Although esoteric and difficult to understand, it is relatively translatable because source code more closely resembles a natural language. The simplest example of executable source code for most programming languages is the "Hello World" program, often used as a beginning lesson in programming textbooks and tutorials. "Hello World" in the C programming language looks like the following:

```
#include <stdio.h>
int main( ){
   printf("hello, world\n");
   return 0;
}
```

Although the meaning of the commands evades non-programmers, it takes little effort to translate: within a main function (int main) it prints (printf) the sentence "Hello World" plus a line break ("\n") to the screen. This is, for all intents and purposes, a complete program. Likewise, *Slash Goggles* (Figure 1) is a representation of source code.

This code, although dense and probably confusing to most, still provides a foundation for "reading" some of the intended meaning of the author. Unlike "Hello World," *Slash Goggles* is "pseudocode," or nonexecutable text written as code in order to map the logic of an algorithm. Pseudocode represents a sort of conceptual, nonexecutable "draft" version of a future piece of code.

Slash Goggles

Russo posted *Slash Goggles* to her Livejournal blog *The Archive2*, described by Russo as "experimental fan works for *Battlestar Galactica* season 4 and beyond,"[11] The blog organizes its contents around a shared interest in *Galactica* and the identity-challenging themes present in the show. *Slash Goggles* addresses *Galactica's* themes directly through the conceit of the Cylon—human-like cyborgs who experience emotions like pain, fear, and sexual desire, and who inevitably model their identities upon (and serve as model for) their human counterparts. The blog post ("the *Slash Goggles* algorithm") contains the *Slash Goggles* code, a description of that code, and a series of images from the series, photoshopped with thought bubbles that reflect what the outcome of the algorithm is. As such, Russo's fan fiction more closely resembles a documentation

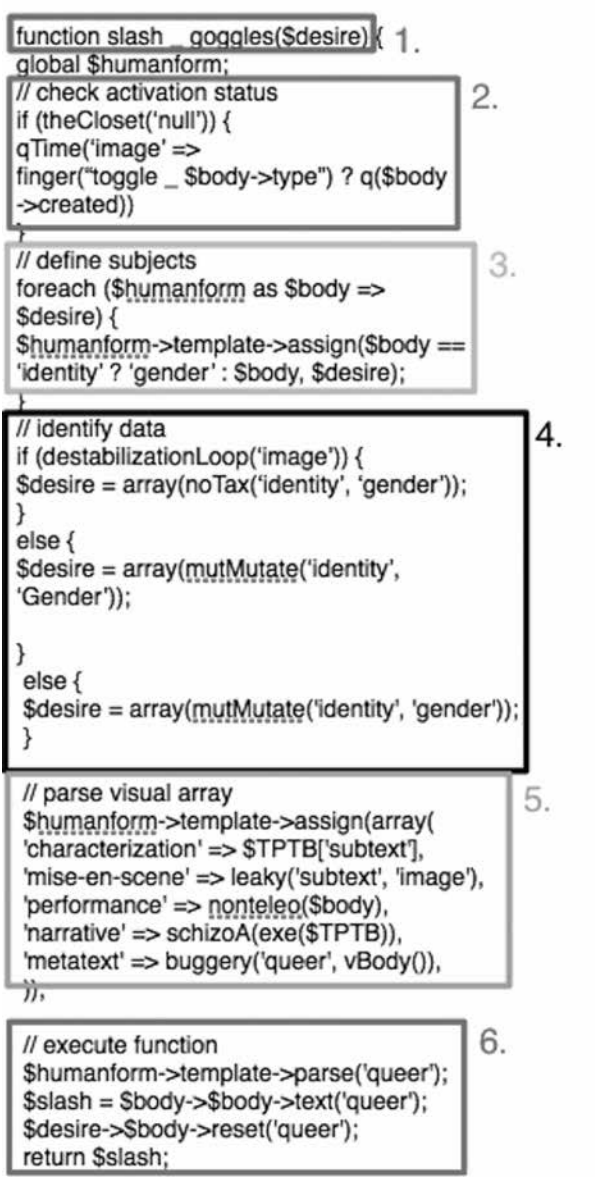
function slash _ goggles($desire){ 1.
global $humanform;
// check activation status
if (theCloset('null')) {
qTime('image' =>
finger("toggle _ $body->type") ? q($body
->created))
}
2.
// define subjects
foreach ($humanform as $body =>
$desire) {
$humanform->template->assign($body ==
'identity' ? 'gender' : $body, $desire);
}
3.
// identify data
if (destabilizationLoop('image')) {
$desire = array(noTax('identity', 'gender'));
}
else {
$desire = array(mutMutate('identity',
'Gender'));
}
else {
$desire = array(mutMutate('identity', 'gender'));
}
4.
// parse visual array
$humanform->template->assign(array(
'characterization' => $TPTB['subtext'],
'mise-en-scene' => leaky('subtext', 'image'),
'performance' => nonteleo($body),
'narrative' => schizoA(exe($TPTB)),
'metatext' => buggery('queer', vBody()),
));
5.
// execute function
$humanform->template->parse('queer');
$slash = $body->$body->text('queer');
$desire->$body->reset('queer');
return $slash;
6.

Figure 1

page for software than a fictional narrative, including descriptions of how the code works and an implementation guide. This documentation describes *Slash Goggles* as a program that "establishes, based on visual data, the variant sexual preferences of human subjects," a program that helps Cylons understand human sexual orientation and "improve HCI" (human-computer interaction).[12]

The fictional language *Slash Goggles* portrays (Cylon Core Language—CyCL) functionally and textually represents scripting languages like PHP or JavaScript. It starts with (1) a function declaration ("function slash_goggles($desire)") that signals the beginning of the algorithm by way of naming it. Functions are, much like basic algebra, names that "stand in" for larger computational formulas that involve variables. In this case, "slash_goggles" names the function, and defines what variable data it takes as input ($desire). Words beginning with a dollar sign ('$') designate variables. Like math problems, variables take a value such as a number or a string of words assigned by the program to do work with or on that value. Following that, a series of subprocedures accomplish component operations: checking activation status (2), defining subjects (3), verifying data (4), parsing that data (5), and using that data to computer some sort of gender identification (6). *Slash Goggles* takes as input the variable "$desire," which immediately implicates some sort of value that quantifies or represents input that has been assigned to represent "desire" by some other program. Then, in a series of smaller subprocedures, the algorithm accomplishes a set of tasks seemingly integral to the identification of sexuality. The "exactness" of code in expressing a procedure through concrete statements and quantifiable results demands a specificity that *Slash Goggles* both leans upon for its rhetorical effectiveness and, at the same time distorts, through linguistic ambiguity.

Slash Goggles critiques and destabilizes both gender and computation as purely mechanical or determined categories, and questions the distinction between the two by staging sexual identification as a procedure. Because the code represents a procedure, Russo also includes some fictional "output" of the code. Images from the show featuring human characters, seemingly viewed through the point of view of a Cylon, are given thought bubbles that articulate that character's sexual desires, frustrations, and complexities. Figures 2 and 3 represent the results of the algorithm as the Cylon characters (in first-person point of view) gaze upon the human characters.

Within the framework of her fan fiction, Russo suggests that this algorithm is similar to human recognition of sexuality ("gaydar"), and explains that *Slash Goggles* represents the procedural work of ascertaining a shifting, fluid sexuality that is drawn from, but not determined by, social dynamics and negotiated subjectivities.

The *Slash Goggles* operation/algorithm describes the negotiation of sexual identification. Here, we can start to look for textual clues to interpret what *Slash*

Figure 2

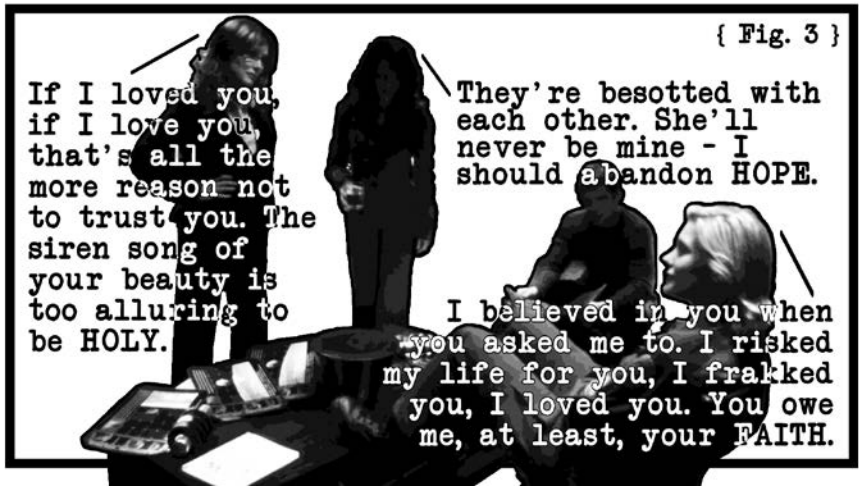

Figure 3

Goggles attempts to express as a critique, because the ambiguity between the language of the code and its use as code helps constructs the potential meanings of *Slash Goggles*. Readers unfamiliar with code may still pick up on references to sexuality, gender, and queer epistemologies in phrases like "theCloset," "buggery()" and references to the body, identity, and gender. Those familiar with coding will also notice the programming structures that meaningfully situate

those references. For example, in "defining subjects" (Section 3 in Figure 1), *Slash Goggles* iterates over each element in $humanform (which, as suggested by the code, contains at least one, if not more, discrete elements) and assigns a value based on visual data ascribed to a variable "$body."

The rest of the algorithm follows suit, and calls to functions that reference "theCloset," destabilization, and performance suggest a similar reliance on contextual, yet open and circulating, interpretations of subjects and their sexualities. As Eve Sedgwick argues, the "closet" is "a performance initiated as such by the speech act of silence—not a particular silence, but a silence that accrues particularity by fits and starts, in relation to the discourse that surrounds and differentially constitutes it."[13] The use of "theCloset" as a code term suggests a self-orientation (is the Cylon subject "in" the closet? Can the Cylon perform or express desire? How does this orient the Cylon towards the subject of their view?) and a cultural orientation constructed by the ability to speak or the necessity of remaining silent and/or hidden. More directly, "theCloset" plays a part in construction of sexual knowledge for the Cylon—if "theCloset" is "null," meaning without value, then the algorithm begins openly constructing data based on sexual traits gained from interpretation of the "$body." The computation of a sexual identification of the other is already based on the viewer's interpretation of the other, which is inherently incomplete and limited, but informed by the available (and, conversely, the unavailable) information.

The algorithm queers sexual identity and identification as well as computation, both in its language and its formal structure. The wording of commands as recognizable ideas in gender and queer theory are mobilized within programmatic constructs like conditional statements, iterating loops, and variables to express a structural critique of sexual identification. By taking a seemingly determined media format (computer code) and using it to demonstrate an ambiguous process, Russo suggests that it simply is not enough to say "sexual identification is ambiguous" or that "gender is dynamic and interpreted," but instead calls for the reader to walk through the difficulty of operationalizing such a task. As Erin Davis writes, "identification occurs within a social regime of normative expectations and guidelines that shape everyone's possibilities for self-representation," and that gender identity is not "static, but it is also not unbounded."[14] In this case, identification is bound by structure and syntax. The only significance between this binding structure (software) and language more broadly is that we have determined a computational, performative context as artificial and knowable. And yet, potential identification is implicated in logical structures and interconnected activities not entirely expressible through procedural understanding. Values, epistemologies, foundations of activity, and knowledge are all continually (re)expressed in these code relationships, even

as they are hidden. The expression and critique of sexual identification is linked, through the use of the algorithm and code as medium for such expression, to a resistance to any notion of an algorithmic reduction of sexual identification as a process and to the reduction of a process as something rote or determined. The underlying performativity connects these approaches as critiques of code and sexual identification, with shared concepts of determination and agency. The connection between these two performative moves is linked through the underlying performative nature of the computational system, and in particular through the gendered nature of that system.

TransCoder and Abstraction

Much as Russo situates identity and identification as a procedure, the underlying work of code implies a realm of computation and execution in excess of a code's meaning. We must therefore situate code as a performative structure outside of strict interpretations of author-intended meaning. Taking source code as the origin or center of meaning, according to Wendy Chun, erases the distinction between code and execution, conflating the meaning of that code with its effects in computer execution.[15] This represents how, according to Chun, how source code is "fetishized" by scholars as the seat of meaning.[16] To fetishize is to erroneously assume that the full meaning of the code is exposed as the source is exposed. Performative code, however, places something like "Hello World" or *Slash Goggles* outside its author as the seat of meaning. As a given piece of software is most commonly tens of thousands (if not millions) of lines of code, the functioning of that software is predicated not simply on a given algorithm, but on the relationships between algorithms in different code libraries and how user input circulates between their execution. If execution and code are separate and under erasure, as Chun argues, meaning cannot be invested strictly in the source code, but within the erasure of the underlying execution of that code—which includes the hiding of hardware and the hiding of underlying code libraries and algorithms. The technique of hiding execution is known in computer science as "abstraction," a method where complex computational processes are hidden or "abstracted" from users behind simpler interfaces to facilitate easier, more streamlined programming.

Abstraction is the mechanism of computational performativity, because this erasure allows the development of complex software from already existing complexity. A good example of this is the software development kit (SDK). An SDK provides programmers with code interfaces that allow them to produce software for complex platforms. A good example of this is the development

of software for mobile devices like smart phones. Android and Apple iPhones are incredibly complex machines, and the code necessary to do even the simplest task (e.g., draw a window to a screen, or connect to the Internet) would be monstrous if each app on that platform required programmers to do each from scratch for every program. Therefore, these platforms have dedicated SDKs, written in a given programming language (Java for Android, Objective C for iPhone) that hides the complexity of hardware and software interaction so that better, more complex apps can be built faster and more reliably. The code required to have an Android phone draw a window to the monitor requires mathematical computations to determine size and orientation, contents, and the very context from which the window comes into being and presents itself to the user (e.g., where to store this in memory, how to toggle between user contexts). When every task requires hundreds, if not thousands, of procedures, suddenly a simple program becomes almost unthinkably complex. However, with the SDK, the command to create a window and start an app is reduced to about a dozen lines of code, none of which require an app developer to manually manage hardware, data resources, or the operating system. Thus, an SDK is an abstraction of the underlying complexity of the phone that allows for the quick and reliable deployment of software for that phone.

Abstraction points us towards the performative because it incorporates regulatory practices outside of the code that we write, and these regulatory practices often consist of assumptions about how people work and communicate. In "The Performativity of Code," Adrian Mackenzie argues that software can represent "a form of collective agency in the process of constituting itself."[17] Using the operating system Linux as his example, Mackenzie writes that the "ongoing constitution is performative with respect to the efficacy of Linux as a technical object and with respect to the fabrication of Linux as a cultural identity."[18] For Mackenzie, Linux regulates its own constitution through a series of norms of practices that are within its own code, which then circulate through the social body of developers that utilize it. This echoes Butler's argument that performativity is a production of subjectivity and sex *through* the iterative power of discourse.[19] That is, it is not an act or performance accomplished by some actor, but the continued (re)production of norms, rules, logics, a reproduction that becomes naturalized or seemingly necessary as it materializes and rematerializes itself. Software "would have to be understood not just in terms of the meanings ascribed to it, or in terms of its effects on the movements of data and information in communication networks. Rather, it would be an objectification of a linguistic praxis."[20] Thus, software must contain within itself the logic of its own reproduction. However, to stay close to Butler rather than, say, Derrida,

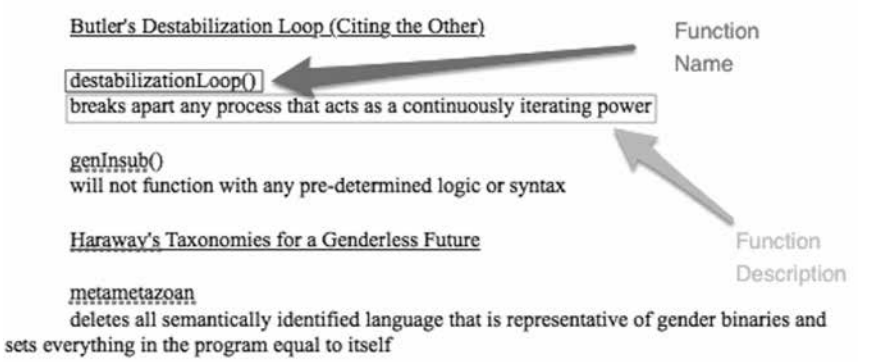

Figure 4

I turn to an example of abstraction that retains Butler's investment in gender: Blas's *transCoder.*

Burned and distributed on CD-ROMS or as packaged files available for download through Zach Blas's Queer Technologies website, the *transCoder* package appears a simple collection of unordered .txt files with names like "about.txt," "libraries.txt," and "compiler.txt." *transCoder,* like *Slash Goggles,* is pseudo code, but whereas *Slash Goggles* mimics an algorithm, *transCoder* more closely resembles an SDK. Marino describes *transCoder* as a "provocative kit" that "uploads counter-cultural ontologies (or anti-ontologies) into the normalized logic of software. He is transcoding theory into a programming language."[21] The accuracy and importance of this description is evident in that, unlike the source code of *Slash Goggles*, what we see here is not an algorithm. In fact, without some prior knowledge of how code functions, we might not have any clue what we are looking at, outside of some recognizable names and references. In the above example, Blas structures a Butlerian concept of iteration and power through the programmatic statement "destabilizationLoop()," and a play on speech act theory with a series of "executable speech acts" (such as "iDo()" and "exe()"). *transCoder*, as an SDK, does not provide a series of code statements that represent the procedures underlying the command "destabilizationLoop()." Instead, the user is given a short description on how "destabilizationLoop()" *should* be used. Or, it tells the user what it does, and not how it does it. "destabilizationLoop()" "breaks apart any process that acts as a continuously iterating power," which implies Butler's assertion that gender and sex are products of regulatory regimes but, through changes during the iteration of such regulative norms (or loops), resistance to such regimes can emerge.[22] The ambiguity of meaning, however, emerges from *transCoder* because meaning is only "implied," because there is no actual code, only the promise of code functionality, because the underlying complexity has

already been abstracted from the user. *transCoder* doesn't define an operation, but the potential foundation for the building of operations, a syntax for further operational complexity, out of the interface of promised functionality. So rather than see the work of code as the process of negotiating meaning between a mind and a machine, between language and hardware, we instead see a performativity of code through logical structure.

Calling back to *Slash Goggles,* Russo utilizes "destabilizationLoop()" in a programmatic construct meant to control the flow of a program through decision making:

```
If (destabilizationLoop('image')){
   $desire = array(mutMutate('identity', 'gender'));
}
```

Because "destabilizationLoop()" breaks apart any process as an iterating power, the use of an "image" suggests the undermining of stable gender categories constructed through visual interpretation. This code snippet uses the "if" statement to check the results of "destabilizationLoop()." If it returns a "true," which seems to occur if the loop actually disrupts the iterating power of the image, then the variable "$desire" gains a series of values resultant from the "mutMutate('identity', 'gender');" statement. "mutMutate()" can "connect any number of items to generate hybrid functions, operators, variables, etc.," and because of this, once the iterating power of "image" is broken, then "$desire" contains the possible hybrid connections of identity and gender. All of the operations described above are rendered meaningful through the framework of *transCoder*, in that the use of *transCoder*'s code calls upon a preexisting coding context. The fact that we know nothing about how "destabilizationLoop()" actually breaks an iterating power demonstrates how my reading can extrapolate meaning from *Slash Goggles,* and how something as seemingly ambiguous as "destabilizationLoop()" still performs within the context of computer code. There is no necessary revelation of *transCoder*'s "code," but only the performative structure that *transCoder* creates.

Also note here a few additional lines from *transCoder*'s libraries.txt. file:

```
schizoA()
Replicates exponentially and erratically

buggery()
acts upon a function or data set and generates an array of monstrous
   non-logic mutations.[23]
```

Both of these functions are found in *Slash Goggles*. Russo, using the "schizoA()" and "buggery()" functions, fills in some gaps as to how each functions works

(e.g., variables, input) by deploying them in a particular way (allocating value, passing variables as input), but their place within a program is articulated through *transCoder*. When *Slash Goggles* calls "'metatext' => buggery('queer,' vBody());" as a command, we may now infer from the documentation that the quality "metatext" takes as its value the result of the "buggery()" function when given the inputs "queer" and the result of another function, "vBody()" (which "detonates a time bomb of radical impurity").[24] The result is that "metatext" will contain some value gained through the process of generating nonlogics of identitarian nonpurity related to queer sexuality, which will then become part of a data structure used to understand sexuality—in this case, the object "$humanform." The movement of structure, meaning, and subject articulation is a product of the citation and reiteration of norms laid out in the library. However, unlike a natural language, the abstraction of complexity, and not the language itself, that serves as the performative engine.

As Chun writes, scholars may fetishize their analyses of source code as the central artifact of meaning, one with a relatively transparent relationship to the underlying machine.[25] Coding qua writing thus becomes a translatable practice between the goals of machine execution and audience persuasion. Although this may not be untrue—in that coding is, in this paradigm, writing—it places the impetus of the practice of coding within a realm of what we would traditionally describe as writing in a professional setting or a classroom: through audiences, intentions, and shared language as a directed activity between writing subject and technology. Abstraction problematizes this while at the same time iterating social norms, including sexuality and gender, because abstraction illustrates code as a logical interface between contexts of complexity. These interfaces are code libraries, like *transCoder*: they simplify the use of complex code so that programmers can build more complex structures. Chun writes that "abstraction both empowers the programmer and insists on his/her ignorance . . . abstraction gives programmers new creative abilities."[26] The work of a programmer is therefore mediated through a series of logical translations across digital media, and this mediation is more or less hidden from the programmer at the site of the production of code. Abstraction through interfaces between contexts of complexity produce programming subject positions and articulates a power structure iterating a particular norm of software production. *TransCoder* and *Slash Goggles* inject queer sexual identification into the performative mechanism to circulate queer identities and world making practices into code itself. Such an injection produces productive discursive strategies for digital rhetoric and queer critique of code.

))) The Queer Performativity of Code and Subverting Mastery

What becomes evident here is that the productive mechanism of computational performativity is no more determined by rote and determined technology than gender by rote and determined biology. In regards to gender performativity, Kendell Gerdes writes that, "the subject of gender is not in charge, but exposed, addressed by the performative power of essentialist understanding of gender rather than the addresser of it . . . the performative power of gender is its ceaseless materialization of gender," and I would argue that in computational performativity, the same could be said about the subject of technology.[27] In this sense, not only are there constraints to performativity but rather, "constraint calls to be rethought as the very condition of performativity."[28] The subject of computation is addressed by the performative power of code through a continual reference to something else—some structure, some language, some order, that it manifests through. There is no stable "machinic subject" in place as we perform identities in online social networks, through medical databases, and so on. There is only the constant production of those subjectivities, and their transformation across various performative machines. In this way, it avoids the presumption of a metaphysics of the subject where there is a "stable gender in place and intact prior to the expressions and activities that we understand as gendered expressions and activities."[29]

From this, a compelling overlap between computational and gender performativities emerges. Both Chun and Mackenzie draw from Butler to argue towards a performativity of code. Both, however, do so more focused on the performativity rather than the gender. If computational performativity is the citation and iteration of complexity across different programming contexts, then I argue that a masculine epistemology of mastery is structurally embedded within code. Drawing from Chun's account of the history of the ENIAC girls, a parallel, gendered rhetoric of mastery with computational performativity shows how subjects are formed in programming and software. Leveraging a discourse of mastery within software development links code to what J. Jack Halberstam calls "queer art of failure," a resistance to the determined and correct (masterful) narratives of computer programming. *Slash Goggles,* as an articulation of *transCoder's* logic, therefore suggests that its proceduralizing of sexual identification is also an articulation and critique of mastery as an epistemology.

According to Chun, the earliest days of contemporary programming involved a large, room-sized computer, the Electronic Numerical Integrator and Computer (ENIAC), a military project developed to help calculate artillery tables

and formulas related to the development of thermonuclear weapons during World War II. Built in the electrical engineering department of Pennsylvania State University, the computer did not use software. In fact, there was no such thing *as* software. Instead, ENIAC programming was the direct, physical programming of the wires, switches, and other components of the machine itself. There was no "code" distinct from the bare metal of the technology itself. The "computers" were a group of women known as the "ENIAC girls," female engineers who operated the machine directly. These women understood the hardware of the ENIAC and how to instantiate solutions to problems given to them by (male) mathematicians and scientists in processes that could take days or weeks. The objectification of these women under a normative understanding of knowledge and discipline continued through their orientation as extensions of that machine. The kind of machine work the ENIAC girls performed was seen as, itself, rote and mechanical rather than creative of intellectual, and such work was often seen as mindless, unskilled, or "clerical." However, after the invention of "software" and code as we know them today, there was an epistemological collapse of the work of programming and of problem posing and solving, and the image of the lone, brilliant computer programmer came into existence. Chun writes that "programming became programming and software became software when command shifted from commanding a 'girl' to commanding a machine," and following this the model constructed around "commanding a girl" became a foundation for machine control as well.[30] As it became abundantly clear that programming itself was a skill worth studying, and as men began to apply scientific epistemologies to describe software and programming as practices, then the programmer became a figure of knowledge, expertise, and mastery. No more a cleric of repetitious activities, the programmer was a priest of a "black art," as John Backus famously described, in which experts handled the arcane through self-driven brilliance and know-how.[31] Nathan Ensmenger argues that programming therefore became masculine when it moved toward categorization as the individual practice of the intellectual "artisan" or the rigorous "scientist."[32]

It is here, in the mistaken assumption that commands may translate directly into machinic code, that the gendered notion of mastery articulates as abstraction and computational performativity. The erasure of complexity is productive in that, in order to produce the conditions of a mastering discursive subject, it erases the invisible and collaborative labor of encoding, translation, and automation in software. Mastery, in this case, is rendered masculine because of its historical nature and because of its epistemological assumptions—the productivity of individual realization through erasure of shared labor. Mastery is the erasure of the intermediary, the investment of creative and productive authority

in the articulated individual of code. Abstraction as a technique *does* ease development for programmers, constructing that discursive subject of mastery, but does so through a willful erasure of the limits and debts of such mastery. Chun notes that "the handing over of power that has been hidden by programming languages that obscure the machine and highlight programming" situates a "master" as an individual, a master not just of the technology but of the epistemology of that technology. The narrative of mastery is predicated on the strategic abstraction of information to constitute agency within another context. In this case, is constitutes a position of agency in which a system (the programming language) is mastered.

Blas argues that *transCoder* "disidentifies" with technology in that it introduces queer critical theory into a seemingly nonqueer space, modifying the circulation of information and identity.[33] Textually, procedurally, it does just this, and in a way that is seemingly apparent to a more general (i.e., nontechnical) audience. Disidentification, however, is a performative act that signals a relationship between an other-ed practice and the larger social logics that marginalize it. For *transCoder* to disidentify means that *transCoder*

> scrambles and reconstructs the encoded message of a cultural text in a fashion that both exposes the encoded message's universalizing and exclusionary machinations and re-circuits its workings to account for, include, and empower minority identities and identifications. Thus, disidentification is a step further than cracking open the code of the majority; it proceeds to use this code as raw material for representing a disempowered politics or positionality that has been rendered unthinkable by the dominant culture.[34]

The inclusion of queer theory as a base for the syntax of *transCoder* signals a complementary rhetorical efficacy rooted in the relationship of the language and its larger, technical–performative context. By utilizing abstraction (and software engineering more broadly) itself as a tactic, rather than as a bracketed epistemology, both introduce into the mastered (or "master-able") space of software the circular, problematic nature of sexual identification, which further, as a performative result, raises and owns the potential and necessity of the failure of that mastery. For Halberstam, failure is located within hegemonic systems of normativity avenues of identification and subversion.[35] In an arena like computer programming, incorrectness or failed code is seen as doomed from the get go—it simply crashes and burns. But if failure is a productive performance through contexts of complexity, rather than in the tight confines of the execution of an algorithm, then failure as a practice can encompass a wider critique of masculine culture in programming. Accordingly, one of Halberstam's theses of failure is that we must "resist mastery."[36] Resisting mastery does not mean simply writing

failed algorithms, but writing structural critiques through performative mechanisms that circulate confusion and drives code logics to illogical ends.

Resisting mastery here means resisting the disciplining urge to reproduce, to correct, to maintain that compels code workers in corporate and academic contexts. Code or software as illogical seems like a contradiction, but only so when the meaning of a given piece of code is invested in the correctness of an algorithm. Bugs, memory leaks, security breaches are all results of the failure of code that escapes correctness, because they all demonstrate the inherent fallibility of the code that always already exists. They are the artifacts of code that escapes its epistemological telos of mastery, which means recognizing the frayed edges of failure that lurks in code inherently, and where this undermines and complicates mastery. Take the "Hello World" example used earlier. The "printf" statement stands as the only real "command" outside of the program structure and header. Printf literally prints a string of characters to the screen, with that string given to the command as an argument. Because it does this, many programmers embed printf statements into their code for debugging purposes, printing warning messages or data values in order to trace the workings of the program. The implication for the programmer, and more often the new programmer, is that printf will "just print" to the output screen when that line of code executes. New programmers often find out that this is not the case, as development variables and errors in compilation can stall out printf so that it prints at the wrong time or not at all. That is because printf actually works through an output buffer system, were the string of characters are placed in memory until a specific symbol is reached ("\n"), which then "flushes" (empties) the buffer and puts the data on screen. StackExchange user Toby Speight found this out during an entry level C programming assignment.[37] His example illustrates how failure is part of coding. His program, which calls the printf command, only "printed" at odd times, or not at all, and certainly not as expected. The culprit, in this case, was in the way his development environment compiled the command and executed it—specifically, in how it handled the buffering and flushing of the data itself. As individuals began to offer solutions on the website, the actual underpinnings of the simple printf command were actually incredibly complex. One user "kliteyn" suggested that Toby try to reset the output buffer with the following command:

```
setvbuf (stdout, NULL, _IONBF, 0);
```

Although another user "Kitchi" suggested switching output buffers more generally, using the "stderr" (error) output command. Another solution suggested, and one prominent in other areas that I have researched, is the "fflush" command, which immediately flushes all output buffers regardless of what else is happening.

Even simple scripts like Toby's or the "Hello World" program are always already built on an ocean of complex code, libraries through which data and system states are passed, transformed, worked on, and returned. Thinking of code through queer failure thus recognizes the avenues by which "failure" is less the breakdown of a program or a piece of software, and more a path of subverting or disidentifying with the disciplinary structure of programming through programming. In the above case, it recognizes the tenuous relationship of "mastery" in the face of such complexity, and the productive constitution of mastery as it breaks down and reforms under a necessary admission of ignorance and need for collaboration. Understanding a command like printf as a command for a machine hides the machinic work that makes printf function and, at the same time, the fluid nature in which libraries, compilation, and execution exceed and problematize the completeness of that understanding. The abstraction of the language, of the underlying complexity, therefore constructs a discursive subject that on the one hand can practice a supposed or assumed mastery of code while at the same time including the necessity of collaboration, of consultation. The parallels between the pseudocode examples of *transCoder, Slash Goggles,* and "Hello World" are such that the ghosts of gendered labor and gendered identification in programming epistemologies ask for different, sometimes complementary and sometimes contradictory, approaches. Primarily, the exactness of mastery is historically situated as masculine, practiced in code that always cites the abstract structure of its predecessors and reiterates them.

Thus *transCoder* and *Slash Goggles*, outside of their own critique, also offer a critical rhetorical strategy for grappling with gender, sexuality, and code by directly injecting it into the performative apparatus of software development. If we trace how code abstracts and separates contexts of complexity from each other, how it functions as an epistemological performance, then the possibilities for understanding software as a gendered practice are more apparent to us. This is why Ensmenger's claim that coding became a boy's club is so relevant. For Ensmenger, the building of masculine epistemologies around the practice of code marginalized women out of the field: it required more work and more overhead in terms of learning style. The representation of women in programming has always been a problem, in particular in places where gender-neutral narratives of achievement, merit, and expertise are stronger (many open source development projects fall under this category).

Butler writes that it is this constitutive failure of the performative, this slippage between "discursive command and its appropriated effect" that "provides the linguistic occasion and index for a consequential disobedience."[38] Or, as Sara Ahmed argues, the demands and prohibitions "of fields or grounds for action are generative: it is not that bodies, objects, inhabit structure, but that bodies are

expressed, inflected, and oriented by their grounding."[39] Blas's move to define an SDK that abstracts complexity pulls us away from trying to figure out the "code" that makes it all work. This same abstraction allows the movement of meaning by structuring the ability of programmers to build further complexity through interfaces like an SDK. The performativity of gender and the performativity of digital code share an important aspect: that they constitute logics of circulation, one of gender expression, one of software and information. These logics of circulation do not carry meaning but produce it, generate it and its conditions. In networked space, this also includes how bodies are translated across grounds of action, between and through them. These activities, epistemologies, and practices are seemingly played out in a space that should be more strictly bounded. Computer technology, however, is not, and at some point the erasure of distinction between execution and code grounds itself in the mastery of the totality of computation. But as *Slash Goggles* and *transCoder* show us, and as we further see performed in "Hello World," there is always already a break in the narrative of mastery: the computer evades us as it pulls from and escapes into culture, labor, and history.

))) Conclusion

Practices of coding are not represented strictly in terms of words, statements, or commands, but in a structural relationship between these commands and a space of intelligibility that the programmer works in, which is turn constructed through a series of translations and transpositions across differing computational contexts. It is perhaps unsurprising, then, that the question of how gender functions as a discourse within technology is further complicated with questions about how gendered practices materialize in computer code through different contexts. If gendered practices exist at the level of code structure, and not just in the text of a program itself, then code as a discourse relies on structural relationships that are unique to code, that only partially resemble language. If we choose to bracket certain technological questions such as the structure of code as a practice of building software, it may lead us to miss important realizations about that technology. In an interview with *Rhizome*, Blas stated that the work of the Queer Technologies project (of which *transCoder* is a part) is to explore a "viral" aesthetics. For Blas, "by making and mass-producing 'products,' Queer Technologies is able to exist in a variety of contexts without necessarily being identified as an art object. . . . They are all designed to be collective engagements, to be collectively experimented with."[40] Furthermore, such experiments can help us examine how "heterosexual mathematicians and scientists

create models and technologies that are infused with heterosexuality," and how "homosexual desires can inform and help to materially construct the technicity of objects."[41] Blas demonstrates the necessary transcontextuality of *transCoder* and the fluid nature of its subject matter not only to express gender critique, but to structure platforms for a dynamic critique of computation and sexuality. Even though sexuality and computation seem like disparate topics, Blas looks under the sociocultural manifestations of computer programming, computer culture, and gender to comment on the epistemological practices of computer programming, which are themselves structured by modes of being that include gendered and sexed practices of communicating and producing knowledge. *Slash Goggles* inhabits this space in order to proceduralize sexual identification, exposing the ways in which one is not without the other, if we care to trace the performance of gender—and other categories—systemically across digital media. Through a conglomeration of rhetorical code practices and performative investigations, scholars will begin to see how seemingly disparate texts, materials, or technologies actually co-constitute one another. The critical is more than ever experimental, in the place where experimentation seems the most constrained.

I have argued that computational performativity is gender performativity when viewed through technical and historical contexts of labor and collaboration in technical fields like computer programming. Both Russo and Blas respond to the rhetoric of software engineering, the how of code circulation and performance, to comment on topics of gender construction, sexual identity, and how both play out in digital technology. But, as I have argued, this critique expands into a queer technique of rhetorical engagement with code that recognizes how structure, form, data, and complexity shape and constrain discursive positions in coding as a discipline. Computational performativity, in the manner I have discussed, offers the potential to look for gendered practices across computer code—not as representations of gender through discourse, but as the ordering of media and media consumption as gendered to produce discourse. Although I believe there are numerous implications for the application of such analysis, the primary vision of this argument is to offer an approach that recognizes code as epistemology, epistemology as gendered and performative, and as such the potential to see a continuum of gendered practices across software and its use. Such a vantage, I believe, offers two immediate benefits.

First, scholars of a critically queer code studies can develop a body of work that explores the relationships between gender representations in STEM industries and academia through software—its production, maintenance, and circulation. Because performativity, computational and gendered, is the continued creation of new worlds, a focus on the queering of software at its own game is a critical imperative. Second, as studies in the fields of computer science,

computer engineering, and electrical engineering have already begun to note the skewed representation of cis-gendered masculinity in programming, a productive interdisciplinary framework between STEM fields and studies in queer sexualities can emerge from a shared critical discourse. An increasing number of studies from the fields of computer science and software engineering point towards gender and racial disparities, not just evident in hiring patterns but in how these fields unknowingly invite or exclude certain individuals through their cultural assumptions and teaching and working practices.[42] Although questions of literacy, education, and social attitudes towards gender and engineering all play into these conversations, critical gender scholars can collaborate with these fields to further investigate how these social realities are present and productive in software. Collaboration in this key stands not only as a corrective measure, but as a way to inform how computer scientists produce software, digital subjectivities, and with them our contemporary culture.

NOTES

1. Wendy Chun, *Programmed Visions: Software and Memory* (Cambridge, MA: MIT Press, 2011), 31.
2. The six original ENIAC programmers were Kathleen McNulty, Betty Snyder, Betty Jennings, Marlyn Wescoff, Frances Bilas, and Ruth Lichterman. See W. Barkley Fritz, "The Women of ENIAC," *IEEE Annals of the History of Computing* 18, no. 3 (1996): 15.
3. Judith Butler, *Excitable Speech: A Politics of the Performative* (New York: Routledge, 1997), 28.
4. N. Katherine Hayles, *My Mother Was a Computer: Digital Subjects and Literary Texts* (Chicago: University of Chicago Press, 2005), 54.
5. Ibid., 52.
6. Robert E. Cummings, "Coding with Power: Towards a Rhetoric of Computer Coding and Composition," *Computers and Composition* 23 (2006): 430–443.
7. Mark Marino, "Critical Code Studies," *Electronic Book Review* (Winter 2006), http://www.electronicbookreview.com/thread/electropoetics/codology.
8. Ian Bogost, *Persuasive Games: The Expressive Power of Video Games* (Cambridge, MA: MIT Press, 2007), 29; Kevin Brock, "One Hundred Thousand Billion Processes: Oulipian Computation and the Composition of Digital Cybertexts," *Techno-culture: An Online Journal of Technology and Society* 2 (2012), https://tcjournal.org/drupal/vol2/brock.
9. Judith Butler, *Bodies that Matter: On the Discursive Limits of "Sex"* (New York: Routledge, 1993), xii.
10. Hayles, *My Mother Was a Computer*, 48.

11. Julie Levin Russo, "About," *The Archive*, http://thearchive2.livejournal.com/profile (accessed February 25, 2016).
12. Julie Levin Russo, "Visual Informatics: The *Slash Goggles* Algorithm," April 10, 2008, accessed March 1, 2017, http://thearchive2.livejournal.com/1465.html.
13. Eve Kosofsky Sedgwick, *The Epistemology of the Closet* (Berkeley and Los Angeles: University of California Press, 1990), 4.
14. Erin C. Davis, "Situating 'Fluidity': (Trans)Gender Identification and the Regulation on Gender Diversity," *GLQ: A Journal of Lesbian and Gay Studies* 15, no. 1 (2008): 98–100.
15. Wendy Chun, "On 'Sourcery,' or Code as Fetish," *Configurations* 16, no. 3 (2008): 303.
16. Ibid., 309.
17. Adrian Mackenzie, "The Performativity of Code: Software and Cultures of Circulation," *Theory, Culture, and Society* 22, no. 1 (2005): 73.
18. Ibid.
19. Judith Butler, *Gender Trouble: Feminism and the Subversion of Identity* (New York: Routledge, 1990), 133.
20. Mackenzie, "The Performativity of Code," 76.
21. Mark Marino, "Of Sex, Cylons, and Worms: A Critical Code Study of Heteronormativity," *Leonardo Electronic Almanac* 17, no. 2 (2012): 189.
22. Butler, *Gender Trouble*, 130.
23. Zach Blas, "libraries.txt," *Queer Technologies* (2012), http://www.zachblas.info/projects/queer-technologies (accessed December 5, 2015).
24. Ibid.
25. Chun, *Programmed Visions*, 29.
26. Wendy Chun, "On Software, or the Persistence of Visual Knowledge," *Grey Room* 18 (2004): 38.
27. Kendall Gerdes, "Performativity," *TSQ* 1, nos. 1–2 (2008): 148.
28. Butler, *Bodies that Matter*, 59.
29. Judith Butler, "Performative Agency," *Journal of Cultural Economy* 3, no. 2 (2010): 147.
30. Ibid., 29.
31. Richard Wexlblat, ed., *History of Programming Languages* (New York: Academic Press, 1981): 69.
32. Nathan Ensmenger, "Making Programming Masculine," in *Gender Codes: Why Women are Leaving Computing*, ed. Thomas J. Misa (Hoboken, NJ: Wiley, 2010), 130.
33. Blas, "introduction.txt," *Queer Technologies* (2012).
34. José Esteban Muñoz, *Disidentifications: Queers of Color and the Performance of Politics* (Minneapolis: University of Minnesota Press, 1999), 31.
35. J. Jack Halberstam, *The Queer Art of Failure* (Durham, NC: Duke University Press, 2011), 89.
36. Ibid., 11.

37. See the entire discussion thread at http://stackoverflow.com/questions/13035075/printf-not-printing-on-console. StackExchange is a forum through which users may ask questions, usually technical in nature, and have them answered by others. It works through a karma system where successful or useful answers are given votes, whereas unhelpful answers or repetitive questions are, by the rules of the site, ignored or rerouted.
38. Butler, *Bodies that Matter*, 82.
39. Sara Ahmed, "Orientations: Towards a Queer Phenomenology," *GLQ: A Journal of Lesbian and Gay Studies* 12, no. 4 (2006): 558.
40. Jacob Gaboury, "Interview with Zach Blas," *Rhizome* (August 2010), accessed April 16, 2017, https://rhizome.org/editorial/2010/aug/18/interview-with-zach-blas.
41. Zach Blas and Micha Cárdenas, "Imaginary Computational Systems: Queer Technologies and Transreal Aesthetics," *AI and Society* 28, no. 4 (2013): 561.
42. Although there are hundreds of studies and editorials on the topic of gender and technology, see Jennifer Tsan, Kristy Elizabeth Boyer, and Collin F. Lynch, "How Early Does the CS Gender Gap Emerge? A Study of Collaborative Problem Solving in 5th Grade Computer Science," *SIGCSE '16*, Memphis, TN (2016), 388–93; Fiona McNair, "The Womanly Art of Programming," *Horizons* 13, no. 3 (1999): 7–10; Claudia Herbst, "Then and Now: Gender, Code, and Literacy," *Social Semiotics* 14, no. 3 (2004): 335–48; Janet Carter and Tony Jenkins, "Gender and Programming: What's Going On?," *ITiCSE '99* (July 1999): 1–4; Carter and Jenkins, "Gender Differences in Programming?," *ITiCSE '02* (June 2002): 188–92; and Ronald Dattero and Stuart D. Galup, "Programming Languages and Gender," *Communications of the ACM* 47, no. 1 (2004): 99–102.

)))

Gerald Stephen Jackson is a PhD Candidate at the University of South Carolina. He investigates the intersections of gender, technical communication, and software development through the lenses of queer theory and critical code studies. His work appears in *Computers and Composition*, *JOGLTEP: Journal of Global Literacies, Technologies, and Emerging Pedagogies*, and the upcoming collection *Rhetoric, Writing, and Circulation* from Utah State University Press.