Computational Culture
a journal of software studies

Software Studies, Revisited. A Roundtable on the Software Studies Series at MIT Press

ARTICLE INFORMATION

- **Author(s):** Wendy Hui Kyong Chun, Winnie Soon, Noah Wardrip-Fruin, Jichen Zhu
- **Affiliation(s):** Simon Fraser University, Aarhus University, University of California Santa Cruz, IT University of Copenhagen
- **Publication Date:** 9th May 2022, as pre-print
- **Issue:** 9
- **Citation:** Wendy Hui Kyong Chun, Winnie Soon, Noah Wardrip-Fruin, Jichen Zhu. "Software Studies, Revisited. A Roundtable on the Software Studies Series at MIT Press." *Computational Culture* 9 (9th May 2022, as pre-print). http://computationalculture.net/software-studies-revisited/.

ABSTRACT

The Software Studies series launched in 2009, under the guidance of editors Matthew Fuller, Lev Manovich, and Noah Wardrip-Fruin. For over a decade, the series was dedicated to publishing the best new work that tracks how software is substantially integrated into the processes of contemporary culture and society through the scholarly modes of the humanities and social science, as well as in the software creation/research

modes of computer science, the arts, and design. Important books published under the tenure of Fuller, Manovich, and Wardrip-Fruin include (among others) Nick Montfort et al.'s collaborative treatise on the single line of code, *10 PRINT CHR$ (205.5 + RND (1)); : GOTO 10;*[1] Benjamin Bratton's comprehensive overview of an accidental megastructure, *The Stack;*[2] and Annette Vee's argument for a computational mentality in *Coding Literacy.*[3]

Each book in the series began with the following introduction:

> *Software is deeply woven into contemporary life—economically, culturally, creatively, politically—in manners both obvious and nearly invisible. Yet while much is written about how software is used, and the activities that it supports and shapes, thinking about software itself has remained largely technical for much of its history. Increasingly, however, artists, scientists, engineers, hackers, designers, and scholars in the humanities and social sciences are finding that for the questions they face, and the things they need to build, an expanded understanding of software is necessary. For such understanding they can call upon a strand of texts in the history of computing and new media, they can take part in the rich implicit culture of software, and they also can take part in the development of an emerging, fundamentally transdisciplinary, computational literacy. These provide the foundation for Software Studies.*
>
> *Software Studies uses and develops cultural, theoretical, and practice-oriented approaches to make critical, historical, and experimental accounts of (and interventions via) the objects and processes of software. The field engages and contributes to the research of computer scientists, the work of software designers and engineers, and the creations of software artists. It tracks how software is substantially*

> *integrated into the processes of contemporary culture and society, reformulating processes, ideas, institutions, and cultural objects around their closeness to algorithmic and formal description and action. Software Studies proposes histories of computational cultures and works with the intellectual resources of computing to develop reflexive thinking about its entanglements and possibilities. It does this both in the scholarly modes of the humanities and social sciences and in the software creation/research modes of computer science, the arts, and design.*
>
> *The Software Studies book series, published by the MIT Press, aims to publish the best new work in a critical and experimental field that is at once culturally and technically literate, reflecting the reality of today's software culture.*

In 2021, a new set of editors—Wendy Hui Kyong Chun, Winnie Soon, and Jichen Zhu—have joined Noah Wardrip-Fruin and reshaped the vision for the series. The revamped series will publish books that focus on software as a site of societal and technical power, by responding to the following questions: How do we see, think, consume, and make software? How does software—from algorithmic procedures and machine learning models to free and open-source software programs—shape our everyday lives, cultures, societies, and identities? How can we critically and creatively analyze something that seems so ubiquitous and general—yet is also so specific and technical? How do artists, designers, coders, scholars, hackers, and activists create new spaces to engage computational culture, enriching the understanding of software as a cultural form? We are especially interested in contributions that move beyond broad statements about software and integrate a wide range of disciplines —from mathematics to critical race theory, from software art to queer theory—to understand the social and cultural implications of software. We seek work that addresses the plurality of practice—from participatory design to critical art—and that draws on knowledge from media, visual, game, cultural and literary studies; history; decolonial theory; new materialism; artistic and critical design practices; and electronic literature and narrative. Ultimately, we seek to explore the vast possibilities, histories, relations, and harms that software encompasses.

To officially relaunch the series, we've come together as a roundtable to share our vision, as we revise and respond to the original series. For this piece, each editor has first offered their vision of why software studies is still necessary and then posed a question for the group about the future of the series. Finally, we've responded to these questions to clarify where we see the series going and to invite authors to join our project.

Wendy Hui Kyong Chun Without doubt, the software studies series in 2009 was radical and necessary. Given the ubiquity and importance of software, it is mind-boggling that it took until then for such a series to emerge. Since its launch, there has been a virtual avalanche of books, book series, and journals dedicated to software in its various forms: from platforms to infrastructure, from big data to machine learning. So, why do we still need such a series, and what do we propose to do with it?

Software studies is still so important because software is so nebulous: it exceeds the categories listed in the above categories; it touches and reshapes—and is touched and reshaped by—almost everything. Software studies enables us to think in broad and/or interconnected terms, to move beyond, between, and beside the various layers and programs. This is no accident. As I wrote in *Programmed Visions: Software* *and Memory* (published in this series in 2011)[4] software seems to help us grapple with the Tower of Babel that is new media by allegedly allowing us to see the invisible whole that generates the sensuous parts. To know software has become a form of enlightenment—a way to comprehend an invisible yet powerful whole—and this conception of software grounds its appeal. Software has become a metaphor for the mind, for ideology, and for the economy: cognitive science comprehends the brain/mind in terms of hardware/software; molecular biology conceives of DNA a series of genetic "programs"; and culture itself has been posited as a form of "software" in opposition to nature, which is "hardware."

At the same time, software is, or should be, a notoriously difficult concept—the clarity offered by software as metaphor should make us pause, because software also engenders a profound ignorance. Who really knows what lurks behind our smiling interfaces, behind the objects we click and manipulate? Who completely understands what one's computer is doing at any given moment? Software as metaphor for metaphor troubles the usual functioning of metaphor, that is, the clarification of an unknown concept through a known one. For, if software illuminates an unknown, it does so through an unknowable (software). This paradox—this drive to grasp what we do not know through what we do not entirely understand—does not undermine but rather grounds software's appeal. Its combination of what can be seen and not seen, can be known and not known—its separation of interface from algorithm, of software from hardware—makes it a powerful metaphor for everything we believe is invisible yet generates visible effects, from genetics to the invisible hand of the market, from ideology to culture.

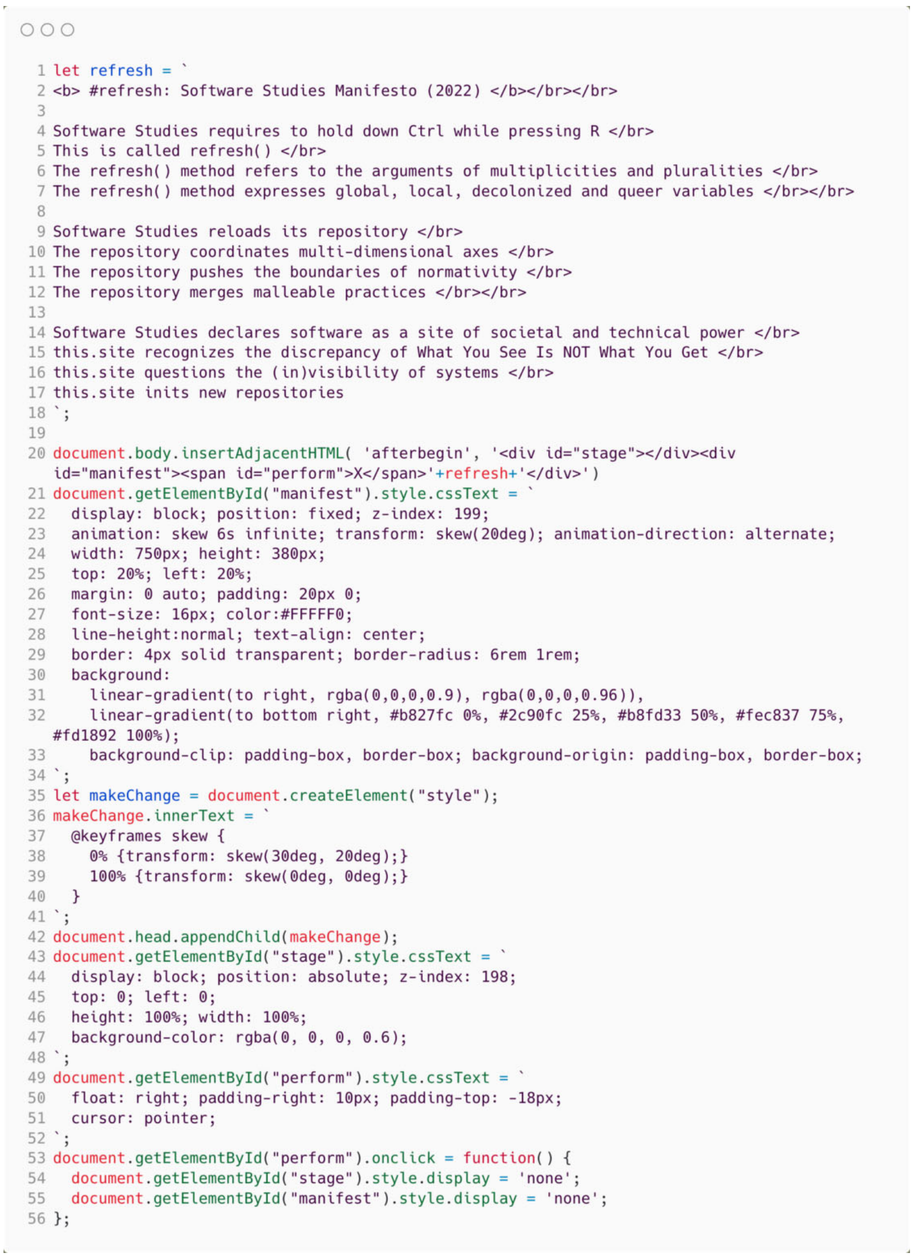
Crucially, software was not planned in advance. Its existence is accidental: the engineers building highspeed calculators in the mid-1940s did not plan or see the need for software. At first, software encompassed everything that was not hardware, such as services. The term *soft* is gendered. Grace Murray Hopper claims that the term *software* was introduced to describe compilers, which she initially called "layettes" for computers; J. Chuan Chu, one of the hardware engineers for the ENIAC, the first working electronic digital computer, called software the "daughter" of Frankenstein (hardware being the son). Software, as a service, was initially priced in terms of labor cost per instruction. Software's emergence as a thing in its own right has everything to do with business models and profound changes to copyright and patent laws, which moved software from something uncopyrightable and unpatentable to something that is both. Business models have again driven its transformation into a service once more. Further, programs existed first in biology and, as I argue in *Programmed Visions*, these failed eugenic visions to program the future informed the emergence of computational programs.

Software's changes thus index profound economic, political, and cultural changes. Historically unforeseen, barely a thing, software's ghostly presence produces and defies apprehension, allowing us to grasp the world through its ungraspable mediation.

The most exciting work in software studies, as evidenced by my fellow editor Winnie Soon's *Aesthetic Programming: A Handbook of Software Studies,*[5] works in and through software's capaciousness and engages work in queer theory, critical race studies, and software arts. As the above description makes clear, software has always dealt with issues of gender, race, and sexuality. As this series "reboots," we will take on these central concerns with work that is as experimental as it is theoretical.

My question to the group thus is: *How do we deal with what's "soft" in software? Again, first defined as everything that was not hardware, software has moved from "layettes" and plug configurations to things to contracts. How can this series address the seemingly chameleon-like nature of software and its relation to changes in gendered- and raced-labor practices?*

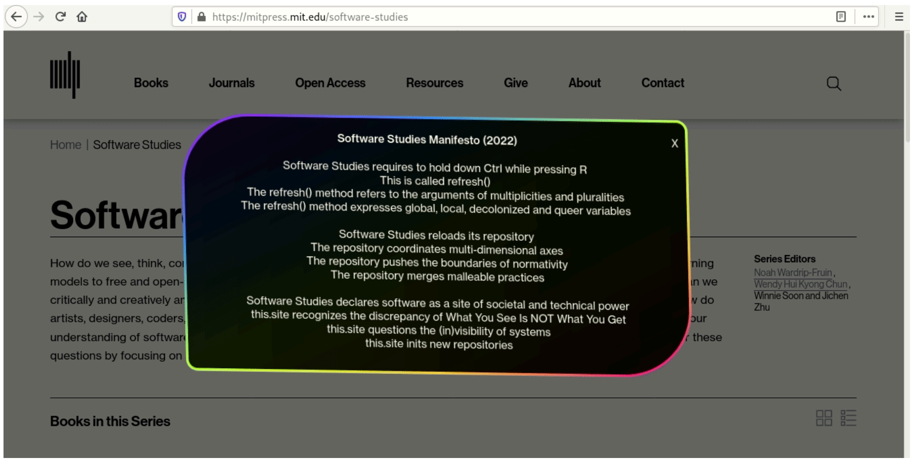
Winnie Soon



```
let refresh = `
<b> #refresh: Software Studies Manifesto (2022) </b></br></br>

Software Studies requires to hold down Ctrl while pressing R </br>
This is called refresh() </br>
The refresh() method refers to the arguments of multiplicities and pluralities </br>
The refresh() method expresses global, local, decolonized and queer variables </br></br>

Software Studies reloads its repository </br>
The repository coordinates multi-dimensional axes </br>
The repository pushes the boundaries of normativity </br>
The repository merges malleable practices </br></br>

Software Studies declares software as a site of societal and technical power </br>
this.site recognizes the discrepancy of What You See Is NOT What You Get </br>
this.site questions the (in)visibility of systems </br>
this.site inits new repositories
`;

document.body.insertAdjacentHTML( 'afterbegin', '<div id="stage"></div><div id="manifest"><span id="perform">X</span>'+refresh+'</div>')
document.getElementById("manifest").style.cssText = `
  display: block; position: fixed; z-index: 199;
  animation: skew 6s infinite; transform: skew(20deg); animation-direction: alternate;
  width: 750px; height: 380px;
  top: 20%; left: 20%;
  margin: 0 auto; padding: 20px 0;
  font-size: 16px; color:#FFFFF0;
  line-height:normal; text-align: center;
  border: 4px solid transparent; border-radius: 6rem 1rem;
  background:
    linear-gradient(to right, rgba(0,0,0,0.9), rgba(0,0,0,0.96)),
    linear-gradient(to bottom right, #b827fc 0%, #2c90fc 25%, #b8fd33 50%, #fec837 75%, #fd1892 100%);
    background-clip: padding-box, border-box; background-origin: padding-box, border-box;
`;
let makeChange = document.createElement("style");
makeChange.innerText = `
  @keyframes skew {
    0% {transform: skew(30deg, 20deg);}
    100% {transform: skew(0deg, 0deg);}
  }
`;
document.head.appendChild(makeChange);
document.getElementById("stage").style.cssText = `
  display: block; position: absolute; z-index: 198;
  top: 0; left: 0;
  height: 100%; width: 100%;
  background-color: rgba(0, 0, 0, 0.6);
`;
document.getElementById("perform").style.cssText = `
  float: right; padding-right: 10px; padding-top: -18px;
  cursor: pointer;
`;
document.getElementById("perform").onclick = function() {
  document.getElementById("stage").style.display = 'none';
  document.getElementById("manifest").style.display = 'none';
};
```

Figure 1: The source code of #refresh: Software Studies Manifesto—refresh.js, 2022.

The above computer source code (refresh.js) is considered a piece of codework, referring to the genre of experimental writing that queers computer and natural languages for both humans and machines to read and perform. Writing a manifesto like this is also writing a piece of software that is expressive, performative, and executable. This draws attention to the argument that code consists of nonneutral commands, not taking for granted any syntax, namings, or even punctuation. Using web-based technologies including HTML, CSS, and JavaScript, web browser extension allows a small piece of software that adds features and functions to a browser. Presenting the work that utilizes the dynamic font type from the browsing site (see figure 2), *#refresh: Software Studies Manifesto* can be read, run, and executed as a browser extension[6] that intervenes in navigating any web pages.



Software Studies Manifesto (2022)

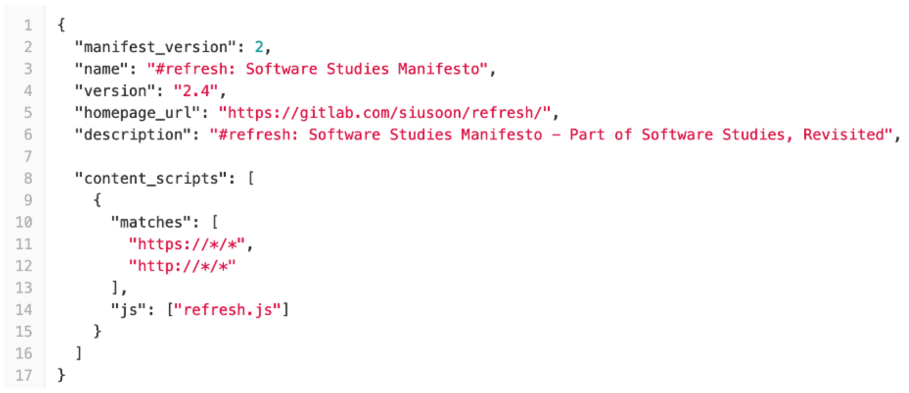
Software Studies requires to hold down Ctrl while pressing R
This is called refresh()
The refresh() method refers to the arguments of multiplicities and pluralities
The refresh() method expresses global, local, decolonized and queer variables

Software Studies reloads its repository
The repository coordinates multi-dimensional axes
The repository pushes the boundaries of normativity
The repository merges malleable practices

Software Studies declares software as a site of societal and technical power
this.site recognizes the discrepancy of What You See Is NOT What You Get
this.site questions the (in)visibility of systems
this.site inits new repositories

Figure 2: #refresh: Software Studies Manifesto, web browser extension (on MIT Press website), 2022.

The piece contains two files: manifest.json (see figure 3) and refresh.js (see figure 1). "manifest.json" is a standardized file name and file format for web extensions, specifying some basic metadata, such as name, version, homepage URL, description, and which JavaScript source files to execute. The naming of this file "manifest" has a specific usage in the area of web technologies, in which it contains startup parameters and information, as a contextualization and a profile, of web applications. Such a web app manifest should be accessible to any web browser or web crawler. As a verb, *manifest* is about "showing or demonstrating something through signs or actions".[7] "refesh.js" involves the content, customization, styling and interaction of the pop up (as a manifesto). As a noun, *manifesto* refers to a written statement declaring and distributing publicly across networks about the intentions and motives of us—we as editors hope to create new spaces to engage with software practices and computational cultures.



```
{
  "manifest_version": 2,
  "name": "#refresh: Software Studies Manifesto",
  "version": "2.4",
  "homepage_url": "https://gitlab.com/siusoon/refresh/",
  "description": "#refresh: Software Studies Manifesto – Part of Software Studies, Revisited",

  "content_scripts": [
    {
      "matches": [
        "https://*/*",
        "http://*/*"
      ],
      "js": ["refresh.js"]
    }
  ]
}
```

Figure 3: The source code of #refresh: Software Studies Manifesto—manifest.json, 2022.

My question to the group is: *What excites you about Software and Software Studies? Since a piece of software can be highly technical, how can we negotiate the space between software technicity and the wider planetary scale of software culture for critical inquiry?*

Jichen Zhu It seems that we have never been better at making software. From augmented reality, deep neural networks, and virtual assistants to blockchains, face ID, and GAN art, software today routinely dazzles users with new capabilities or higher performance than what was barely thinkable only years before. "Intelligent" systems grew "Deep" and "Big." The Internet of Things and non-fungible tokens continue to blur the boundary between the physical and digital, hardware and software.

At the same time, we know increasingly less about the software we make. The software stack necessary to run an application grows thick; each layer obfuscates what is actually happening beneath. Rule-based symbolic artificial intelligence gave way to statistical machine learning models trained from enormous datasets. As a result, even technical experts struggle to fully make sense of these black boxes. Economic interests, rising geopolitical cyber threats, and nationalistic protectionism all exert pressure on software developers to make software ecosystems more closed.

While we can theoretically make software do anything, in practice we become less confident about what to make. New waves of human-computer interaction are shifting away from well-defined communities of practice and the traditional engineering culture it inherited. Researchers remark on the "ontological uncertainties, epistemological diffusion and ethical conundrums" [8] that the field currently faces. For example, focusing only on immediate user satisfaction has contributed to widespread long-term concerns such as the Filter Bubble. How do we evaluate software products to account for longitudinal implications and societal effects? Autonomous cars already actualized the ethical Trolley Problem into a realistic scenario; how should designers of the cars' control software approach these decisions?

What are our desirable technological futures? This is a technical design question as much as a philosophical, ethical, and political one. It requires critical analysis of existing software practices as well as constructive reimaginations of future possibilities.

Coders, UX designers, HCI practitioners, and all makers of software: this is why you need software studies.

The most important question for me is: *How can this book series help to extend insights from software studies to inform critical making and design of software?*

Noah Wardrip-Fruin I believe our best hope for grappling with software is radical intellectual promiscuity.

To understand the specific dangers of algorithmic policing requires delving into software that can be understood only by integrating disciplines such as mathematics and critical race theory. To understand how social networks shape communities, identities, conspiracies, and hate speech, we need to combine tools from areas such as sociology, psychology, and human-computer interaction with tools from areas such as network analysis and artificial intelligence. To understand the possibilities and ideologies embedded in algorithmic media forms, such as computer games and interactive narratives, we must draw on knowledge from areas such as media, visual, and literary studies while also examining the tower of contingent abstractions upon which such media's mechanisms and assumptions are inscribed. To find new ways to make software, we must bring together ways of working, from participatory and critical design to activist and free software communities, with new ways of understanding our projects and evaluating our work—which are more likely queer and decolonial than Taylorist.

The only problem, of course, is that we have organized the world to be inhospitable to such ways of thinking and making. For example, we have a paucity of degree programs, intellectual gatherings, and exhibition and publication venues that would welcome (or even tolerate) such work. My hope is that the reimagined Software Studies series can serve as another home, and support, for those pursuing an understanding of the specifics of software with the undisciplined, eclectic approaches our world needs now.

For me, the most important question is: *How do we find and support such authors, or teams of authors? How do we help shape circumstances so that it is more possible for people, especially junior researchers, to take the time and risks that such work demands?*

***

Questions and Responses

Question: *How do we deal with what's "soft" in software? Again, first defined as everything that was not hardware, software has moved from "layettes" and plug configurations to things to contracts. How can this series address the seemingly chameleon-like nature of software and its relation to changes in gendered- and raced-labor practices?*

Responses

WS: Many historical (or even contemporary) technological categories are gendered and hierarchical. Beyond the division of soft/hard, other categories such as master/slave are clearly not neutral—language or metaphor engenders particular ways of seeing and knowing. The historical and gendered concept of the "soft" in software exemplifies the ways in which cultural and technical conceptions affect one another. "Soft"ware is often linked to procedures, instructions, execution, and programming notations that are not easily "seen." The ENIAC, the first programmable digital computer built in the 1940s, was programmed not by typing symbolic syntax but rather by the physical movements—wiring and switching—of women laborers. Thus, to understand software, we need to address not only the technical procedures of wiring and plugging but also the cultural and social conditions under which software has evolved, developed, and been understood over time.

In terms of the historical evolution of software from a service to a "thing" (and now back to a service), we are now observing a different type of service operating in computational culture. Cloud-based services are commonly used by corporations, in which services are delivered through a network of remote servers and infrastructure. The shift of downloadable and installable software to remote access to software services is changing business models, the role of users, and their relationships with computers. But to return to the issue of gendered- and raced-labor practices, we can also see how the programming profession has been gradually transformed into "a high-status, scientific, and masculine discipline" [9] which is very different from the 1960s where many women entered the tech job market without prior coding experience as they had on-the-job training.[10] Therefore, understanding the conditions of software from historical, gender, societal, and class perspectives is important to our ability to see software as a site of societal and technical power.

NWF: "Software" has a rich life as a metaphor, but we don't have to deal with it only metaphorically. Digging into the specifics of how it operates; of the labor practices (that are raced, classed, and gendered) through which it is created, tested, maintained, and updated; of the ways it is received and interacted with; of the ways it is used within, and substitutes for, legal frameworks; of the ways it exerts power and breaks down—how these ground and exceed our common metaphorical frames—this is work I find immensely important and still too rare.

To put it another (metaphorical) way, software is soft in that it is always taking on new shapes: it flutters in the wind; it moves away when we try to put pressure on it. But that doesn't mean our work should end with observing these aspects of it. We should run our fingers over its texture; we should trace the supply chains, labor practices, and environmental impacts of its manufacture and sale; we should pull and bunch and twist and cut and sew, to reveal more about it and turn it to new ends.

JZ: Indeed, the software industry has relied on metaphors to describe functionality as well as to signal its values. For instance, today's software is often "big" (data) and "deep" (learning). These metaphors can be studied both culturally and technically. For instance, Agre[11] identified that, to achieve "intelligence," classic AI metaphorically connects the vernacular and technical meanings of key terms such as "planning" and "learning." Mateas declared that every software contained an intertwined code machine and a rhetoric machine.[12] What is "soft" in software thus can also be the narratives and interpretations people develop about software's operation.

Underneath the metaphors of the "big" and "deep" is a narrative of what Meredith Broussard has called Technochauvinism.[13] Their technical foundations rely on opaque "neural network" models, industry-scale processors, and enormous data sets. To make sense of these, one needs to investigate issues such as gender, race, and corporate power. Who is creating and cleaning these data sets? What are their working conditions? How much energy do the tensor processing units (TPUs) consume? What is the environmental cost of cooling the data centers?

WHKC: Thank you for these insightful answers, which range from examining the role sexism plays in making things invisible to calls to examine the environmental costs of software services, from the inherent plasticity of software to the impact of metaphor on our conceptions of technology. I think that our collective interests in supply chains, labor, and environmental costs will push the series in new directions and broaden the range of both the readers and authors. I see us publishing more books that bring together technical and social infrastructures, along the lines of Tung Hui Hu's *A Prehistory of the Cloud*,[14] Meredith Broussard's *Artificial Unintelligence* [15], or Mar Hick's *Programmed Inequality*,[16] as well as philosophical ruminations such as Yuk Hui's *Recursivity and Contingency*[17]. I think the biggest gap so far in the series has been books that bring together critical race theory and software studies—issues addressed so insightfully by so many authors, including members of the Center for Critical Race and Digital Studies (Andre Brock, Safiya Noble, Charleton McIlwain, among others).[18] I look forward to working with authors taking on these issues and expanding the scope of software studies.

<u>Question: </u>*What excites you about Software and Software Studies? Since a piece of software can be highly technical, how can we negotiate the space between software technicity and the wider planetary scale of software culture for critical inquiry?*

<u>Responses</u>

WHKC: I think that what excites me most about software and software studies are their incredible breadth—and how they challenge us to think about what's visible (interfaces, content, etc.) and what's invisible (hardware, infrastructures, economics, etc.) at the same time. Software, as I elaborate in my response to the next question, can't be grasped using one method or approach. Negotiating the space between technicity and culture is hard—but it's exactly what we need to do. To do so, we have to realize that no one completely understands software. I'm trained as an engineer, and I can tell you what my computer is doing at many levels theoretically, but I can't tell you exactly what it's doing right now. Even though I'm also trained as a humanist, I don't know what any given human is doing or thinking. These are both enabling gaps in knowledge—and if we obsess too much about what's unknown (as though everything should be transparent and that "seeing through" something means "seeing it truthfully"), then we miss the importance of what's unknowable. For me, we need to bring technical, cultural, and social science methods together so we can map the productive force of the unknowable, and thus try to understand how we construct knowledge and its impacts.

NWF: When you think about making software, one important aspect of its technicity is that you're not always starting over. If you learn to program in Lisp, you don't start over when you learn Scheme. If you learn to program in Java (a horrible idea, BTW), you don't start over when you learn C#. You don't start over in learning how syntax coloring helps, or how IDEs or versioning systems help, or in learning debugging techniques. You don't even start over at the level of building software: you reuse a lot of existing design patterns, libraries, frameworks, game engines, and statistical models.

That is, when you're making software, you can think analogically to transfer the experience, knowledge, and tools from one context to another. Part of what excites me about software studies is that I believe we can do this with a critical understanding of software as well. Through books, journal articles, blog posts, and social media bon mots, we can begin to build a set of understanding that will let people move more quickly and easily to critically examining the next software they encounter.

In doing this, I think it's particularly powerful to work with legible examples. By this I mean work like Minh Hua's and Rita Raley's "Playing with Unicorns: *AI Dungeon* and Citizen NLP."[19] In this case, there's legibility on two levels. First, while there are many examples one could choose for writing critically about large language models (in this case, the GPT series) by selecting a game, with iterative audience input and amusing system-generated output, Hua and Raley can illuminate what is going on in prompt engineering in a more engaging and revealing way than if their examples were the kinds of decontextualized "Look, it generated this!" results that Open AI and other language model creators generally select. Second, their example, *AI Dungeon*,[20] itself is being examined as a kind of legible example, one that allows community exploration and reflection on prompt engineering for such models and what it can reveal about the biases in the data and the contours of the generative space.

In other words, I think at least some of our attention in software studies should be on interactive, audience-oriented works—because their legibility makes them particularly powerful in building up critical literacies about software. These literacies help us to examine such works themselves, which are a pervasive part of culture, as well as at least partially transfer our understanding to new software contexts.

JZ: As software grows increasingly opaque and complex, the space between studying its technicality and investigating its cultural ramifications has grown further. Contemporary deep neural networks have such incredibly low interpretability that even their engineers struggle to make sense of it entirely. If we look at the current discourse around AI, with a few exceptions, the technical community and the social sciences are still looking for common ground to build on. It is hugely exciting to me that software studies research, especially work that actively engages in both software technicality and culture, can provide that missing link.

WS: Thank you for all your responses. It is already exciting to hear your points of view and insights in terms of the need for, and how to do, software studies. One of the shared concerns/interests is the unknowability of software, from concrete technical operations, narrative structures, and audience experiences to wider cultural implications and significance, in which this is clearly relevant to many scholars, users, engineers, programmers, artists, and various communities in the society. Making software and producing knowledge may be similar, where there are many ways to build on and understand them, such as deconstructing and analyzing existing games/software/models, collaborating with social scientists and engineers with mixed methods, and producing critiques and practice-based artifacts, among others. It is important to cultivate the multiplicities of critical and creative approaches in unpacking, conceptualizing, examining, speculating, and making software, a cultural object that is not often apparent to our immediate registers but which has significant effects and consequences. As the editorial collective, we are interested not only in the question of what software is about but also in how to study software and its related communities, organization, culture, and practices.

Question: *How can this book series help to spread the insights from software studies to other communities of practices around software?*

Responses

WS: There have been a lot of makers, coders, and designers, but it’s true I also feel the link is somehow missing regarding software/computational practitioners and the field of software studies. One of the ways to bridge this may require unlearning the assumptions and normative practices in doing and developing software beyond just serving or prioritizing neoliberal demands. This, perhaps, starts with software as a site of playful experimentation and aesthetic/critical inquiry, which involves not just functionally solving problems but also posing them to allow new forms of knowledge to emerge. For me, this question is about fundamentally changing or expanding how we normally approach software, creating new spaces and challenging the way we might make software otherwise.

NWF: I tend to think that engineering and art are pretty similar—they're ways of making things (and of course those things can be conceptual or ephemeral). That's not the internal conception of a lot of engineering institutions. As Winnie suggests, they think engineers are "solving problems" and only incidentally making things along the way to solutions. And despite the spread of software to many disciplines, engineering institutions are still where most people learn to make and think about software. In the institutional engineering view of the field, which many people have internalized, the important questions become (1) how the problems to be solved are defined and (2) how we know when we have a solution, or a "better" solution.

It's in these places I see a lot of dissatisfaction among software makers. The problems have been defined incorrectly at Facebook and YouTube, if they boil down to "How do we maximize engagement, no matter how misleading and hateful?" The metrics have been defined incorrectly in much published engineering research, if they boil down to "I made it more efficient, without ever interrogating why my corporate or military funder wanted it that way."

The questions and incentives have even been defined incorrectly in areas with little funding. For example, faculty and graduate students in my own software research area (interactive narrative) don’t have much incentive to actually make the things we say we’re trying to discover new possibilities for (interactive narratives). Instead, we mostly produce technical frameworks and studies of other people’s work—in part because it’s unclear what problem is being solved by making a complete work. And it’s hard to measure success for something as complex as a meaningful work of fiction. You don’t necessarily get more peer-reviewed publications out of a complete project. The landmarks in our field are almost all experimental interactive narratives, but we have organized things so that most dissertations and major projects in the field do not produce one.

All that said, many people are looking for different conceptions of making software. And many people working to develop such conceptions—such as Winnie’s suggestion that software can be considered a way of posing problems. I think these people are doing software studies. Perhaps the term and the series can help those who are seeking and producing new conceptions to find each other.

WHKC: Your question raises two excellent, more general questions: How do we work together across disciplines? And how do we address the apparent gap between theory and practice?

To work effectively across disciplines, we need to take on an issue that one discipline or group can’t on its own. Software—given its ubiquity, diversity, and multivalence—is clearly such an issue. Some approaches focus on technical structures, while others drill down on its economics; some privilege its visible aspects—interfaces and content—while others claim that what really matters can’t be seen—infrastructure, code, etc. Software studies is so important because it brings together all these approaches because one alone is insufficient. Software affects so many of us: from workers whose actions are tracked and ruthlessly “optimized” to students whose actions are also surveilled via educational technology; from would-be social media micro-stars who see every real-world moment in terms of their feed to migrants who use cell phones to negotiate perilous journeys.

How, though, can we get multiple communities of practice to join in the dialogue? One way is to refuse the divide between theory and practice. Crucially, theory and practice weren’t always opposed. As Wlad Godzich explains, “theory” derives from the Greek *theoria*, a term that described a group of officials whose formal witnessing of an event ensured its official recognition. Its opposite was aesthetics, which was the seeing of women and slaves. Theory from its very beginning was performative or productive, but also troublingly exclusive. To move beyond its elitism, we need to engage both what’s been excluded—aesthetics—and communities that have been excluded as forms of witnessing and event-making. How might we bring together various users and practitioners by focusing on how our engagements with software foster different experiences, narratives, and witnesses?

JZ: Thank you for these insightful responses. A common theme is that software studies are an essential component of critical technical practice. [21] From Winnie’s vision of alternative modes of software making to Noah’s observation of the dominant engineering culture, we see the field of software studies as an intellectual toolset for software makers. It can empower them to identify better problems and explore more socially responsible evaluation metrics. Wendy’s argument highlights software studies as an inherently interdisciplinary field that pushes back the traditional boundary between theory and practice. We can imagine a wide range of software studies projects that bridge different disciplines to inform, question, and expand what we currently think of as software making and software culture. We encourage new work that connects critical theory and practice. As coeditors of this series, we also need to further raise the visibility of software studies and build alliances, especially with relevant communities who may not be fully aware of the exciting work here.

Question: *How do we find and support such authors, or teams of authors? How do we help shape circumstances so that it is more possible for people, especially junior researchers, to take the time and risks that such work demands?*

Responses

NWF: I'm in part thinking of *I AM ERROR,* my favorite book in the Platform Studies series.[22] (It's my favorite because it's not only critically insightful in the way it grapples with technical specifics, it also turns its critical attention to the underlying assumptions of platform studies as a project.) The book came about through early-stage support from the series editors, before Nathan Altice (the author) had definitively settled on a dissertation topic. That support opened the way to a dissertation that was pretty unusual, but the prospect of an MIT Press book coming out of the process produced a lot of goodwill. This support from the editors was a bit of happenstance—it started with Altice giving a conference talk, which one of the other series authors saw, which then led to contact with the editors and the press.

I'm wondering if there are ways beyond happenstance that we can help open the way for software studies work.

WS: One of the important aspects around finding and supporting authors is creating a safe and welcoming space, which is something I have learned from open source communities, especially p5.js.[23] We need to make it explicit that we welcome early-career researchers, as well as people with different ways to queer software beyond Western traditions and big tech normalization to account for other forms of knowledge production.

Software studies is still a field required to think about the specificity of software and how that might differ or connect to other technologies such as platforms, data, and the internet. Accessibility is another issue: what are the entry points for people to start engaging with the field? People may not consider the *Software Studies* book series is something for them. It will require some effort to actively look for, engage, and support emerging researchers to work across disciplines and to push the boundaries of how software could be made or studied differently.

WHKC: This is such a crucial and difficult question. The example you bring up is amazing—but it also raises so many questions about the "professionalization" of graduate students and the dwindling job prospects for them. Writing a dissertation is tough; trying to write a dissertation that will launch an academic career is almost impossible, especially since one never knows in advance where the field will be once you're done. (This is of course true for any book-writing adventure.) Having said this, your question is: what can we as editors do to support emerging scholars in the face of everything I've written above? I think that being there for younger researchers is absolutely key—reaching out to them, encouraging them, and giving them a sense of what impact their work can have. Participating in workshops for PhD students about publishing; attending conference panels with PhD students and reaching out to them afterward about the series; helping PhD students apply for postdocs that will expand their projects and help them see their dissertations as books. These are three of many things we should and will do.

JZ: We need to better support researchers who undertake interdisciplinary work. As we discussed earlier, it is particularly challenging to produce work that can deeply engage the technicality as well as cultural implications of software. It is risky for authors to create such works because it takes time to develop competency in more than one discipline. It takes even longer to reconcile the different value systems and methodological differences between disciplines. Yet, such works are so critically needed in software studies. In addition to Winnie's and Wendy's excellent points of mentoring young researchers and fostering a safe space, raising awareness at the institutional level to recognize and support interdisciplinary work can also help.

NWF: Thanks all for these thoughtful answers. I'm intrigued by Winnie's parallel with community-developed software. Connecting with some of Wendy's thoughts: can we imagine workshops with PhD students that aren't just forums for giving advice and asking questions, but collaborations in which we make something together? There's certainly precedent for collaborative work in the series, such as the ten-author book *10 PRINT CHR$(205.5+RND(1)); : GOTO 10*, [24]which began in the context of online discussion through the Critical Code Studies Working Group. [25]It also reminds me of group collaborations on critical Wikipedia editing interventions, such as those organized by Art+Feminism.[26] And that brings me to Jichen's thoughts: it is difficult and time-consuming to develop cross-disciplinary expertise—and even traditional, disciplinary graduate study is already fraught with dangers such as imposter syndrome. Perhaps frameworks and contexts for collaborative making could also help address this aspect of the challenges of doing software studies.

***

To conclude, it's been more than a decade since the original launch of the Software Studies series. In that time, as software and its influences have pervaded more of the world's cultures and economies, as big software companies have stumbled in their attempts to integrate "ethics" (whether with internal researchers or outside boards) and scrambled to defuse and deflect discrimination lawsuits and unionization, as the humanities and social sciences have struggled with how to address the workings of software and the technocultural and infrastructural assemblages through which it is produced and operates, and as the arts have sought some way to engage constructs such as deep learning and large language models beyond producing promotional images and toys, the need for the work of software studies has only grown. In relaunching the series in a mode that is both broader in its view of software and more pointed in its approaches, the new editorial group looks forward to being both venue and support for the urgent work of software studies.

Bibliography

Agre, Philip E. *Computation and Human Experience*. Cambridge, UK: Cambridge University Press, 1997.

Altice, Nathan. *I Am Error: The Nintendo Family Computer / Entertainment System Platform*. Cambridge: MIT Press, 2015.

Bratton, Benjamin H. *The Stack: On Software and Sovereignty.* Cambridge: MIT Press, 2016.

Broussard, Meredith. *Artificial Unintelligence: How Computers Misunderstand the World*. Cambridge: MIT Press, 2019.

Chun, Wendy Hui Kyong. *Programmed Visions: Software and Memory*. Cambridge: MIT Press, 2013.

Ensmenger, Nathan. "Making Programming Masculine." In *Gender Codes: Why Women are Leaving Computing*, edited by Thomas J. Misa. Hoboken, NJ: John Wiley, 2010.

Frauenberger, Christopher. "Entanglement HCI the next wave?." *ACM Transactions on Computer-Human Interaction (TOCHI)*, no. 27.1 (2019), pp.1-27.

Hicks, Mar. *Programmed Inequality: How Britain Discarded Women Technologists and Lost Its Edge in Computing*. Cambridge: MIT Press, 2018.

Hu, Tung-Hui. *A Prehistory of the Cloud.* Cambridge: MIT Press, 2016.

Hui, Yuk. *Recursivity and Contingency*. London: Rowman & Littlefield International, 2019.

Mandel, Lois. "The Computer Girls." *Cosmopolitan* (April 1967) pp.52-56.

Mateas, Michael. "Expressive AI: A semiotic analysis of machinic affordances." *3rd Conference on Computational Semiotics for Games and New Media*, vol. 58. (2003).

Montfort, Nick, Patsy Baudoin, John Bell, Ian Bogost, Jeremy Douglass, Mark C. Marino, Michael Mateas, Casey Reas, Mark Sample and Noah Vawter. *10 Print Chr$(205. 5+rnd(1))*. Cambridge: MIT Press, 2014.

Raley, Rita and Minh Hua. "Playing With Unicorns: AI Dungeon and Citizen NLP." *Digital Humanities Quarterly*, no. 14.4 (June 2021).

Soon, Winnie, and Geoff Cox. *Aesthetic Programming: A Handbook of Software Studies*. London: Open Humanities Press, 2020.

Vee, Annette. *Coding Literacy: How Computer Programming is Changing Writing*. Cambridge: MIT Press, 2017.

Footnotes

1. Nick Montfort, Patsy Baudoin, John Bell, Ian Bogost, Jeremy Douglass, Mark C. Marino, Michael Mateas, Casey Reas, Mark Sample, and Noah Vawter, *10 PRINT CHR$ (205.5 + RND (1)); : GOTO 10*; Cambridge, MA: The MIT Press, 2012 ↩
2. Benjamin Bratton, *The Stack,* Cambridge, MA: The MIT Press, 2016 ↩
3. Annette Vee, *Coding Literacy,* Cambridge, MA: The MIT Press, 2017 ↩
4. Wendy Hui Kyong Chun, *Programmed Visions: Software and Memory.* Cambridge, MA: The MIT Press, 2011 ↩
5. "Winnie Soon and Geoff Cox, *Aesthetic Programming: A Handbook of Software Studies*, London: Open Humanities Press, 2020. See also, https://aesthetic-programming.net/" ↩
6. Download the web browser extension with instructions here: https://gitlab.com/siusoon/refresh/ ↩
7. *Cambridge Dictionary, Cambridge University Press, https://dictionary.cambridge.org/* ↩

8. Christopher Frauenberger, "Entanglement HCI The next wave?" ACM Transactions on Computer-Human Interaction. vol.27. iss. 1, pp.1-27, 2020. ↩
9. Ensmenger, Nathan. "Making Programming Masculine." In *Gender Codes: Why Women are Leaving Computing*, edited by Thomas J. Misa. Hoboken, NJ: John Wiley, 2010. ↩
10. Lois Mandel, "The Computer Girls." *Cosmopolitan* (April 1967) pp.52-56. ↩
11. Philip Agre, <*Computation and Human Experience*, Cambridge: Cambridge University Press, 1997. ↩
12. Michael Mateas, "Expressive AI: A semiotic analysis of machinic affordances." *3rd Conference on Computational Semiotics for Games and New Media*, vol. 58. (2003). ↩
13. Meredith Broussard, *Artificial Unintelligence: How Computers Misunderstand the World*, Cambridge MA: The MIT Press, 2018. ↩
14. Tung Hui Hu, *A Prehistory of the Cloud*, Cambridge MA: The MIT Press, 2015 ↩
15. Broussard, op cit ↩
16. Mar Hicks, *Programmed Inequality: How Britain Discarded Women Technologists and Lost Its Edge in Computing, Cambridge MA: The MIT Press, 2018.* ↩
17. *Yuk Hui, Recursivity and Contingency, London: Rowman and Littlefield International, 2019* ↩
18. *The Center for Critical Race & Digital Studies, https://criticalracedigitalstudies.com/* ↩
19. *Minh Hua and Rita Raley, "Playing With Unicorns: AI Dungeon and Citizen NLP", Digital Humanities Quarterly, Vol. 14 No.4, 2020 http://digitalhumanities.org/dhq/vol/14/4/000533/000533.html/* ↩
20. *AI Dungeon, https://play.aidungeon.io/* ↩
21. *Philip Agre, op cit* ↩
22. *Nathan Altice, I Am Error: The Nintendo Family Computer / Entertainment System Platform, Cambridge MA: The MIT Press, 2015* ↩
23. *p5.js, https://p5js.org/* ↩

24. *Nick Montford et al, op cit ↩*
25. *Critical Code Studies Working Group, https://wg.criticalcodestudies.com/ ↩*
26. *Art + Feminism, https://artandfeminism.org/ ↩*

COMMENT, ISSUE NINE