Sam de Alfaro

Senior Project, Spring 2024

Supervised by Cyd Cipolla

Dismantle the Computer

An Intervention in Computer Science Education

Computer science education as it currently stands does not adequately prepare students for real-world experience. With the rise of limited AI built on Large Language Models (LLMs, such as ChatGPT), the need grows for programmers who can create efficient programs grounded in social and ethical frameworks. These new programmers should be trained to create code that is beautiful, and that can tell stories in order to directly and consciously interact with human activity. By focusing on an aesthetic attunement while coding, programmers focus on design and architecture, ensuring that their code is readable, efficient, and concise. In this way, programmers could create code unlike what AI models can generate; this code would reflect human creativity, empathy, and context sensitivity. In addition, the demand for programmers who can build a website is diminishing, as AI programs are now able to quickly provide the code for any frontend (and most backend) work needed. Humans, instead, are needed to create fast and efficient code that is grounded in a human attunement to beauty and pleasure. Computer scientists should be able to situate their work in material and political processes to understand how their work contributes to existing systems, and in particular, learn to avoid contributing to systemic racism, sexism, or other forms of oppression (Noble, 2018). As a scientific discipline, computer science should equally aim to contribute to the global pool of scientific knowledge and resources.

Computer science education as it is presently formulated does not take these considerations into account.

Various models of computer science education, such as from the lenses of engineering, math, science, and even art, fail to address the underlying issues in the development of computer science. In particular, educational theorists promote several assumptions about who is pursuing science, and why. For instance, education in programming languages often assumes an understanding of hardware that is necessary for a low-level comprehension of how languages work, pointing to a much greater issue in computer science education: the separation of technical information from its broader context. Even accessibility within computing is an often neglected topic passed on to UX designers rather than being integrated into all levels of education and coding. If one is taught how to code a computer without knowing how a computer works, and more importantly, without ever understanding *what* (and *who*) makes the computer work, then how can we expect computer scientists to be aware of the implications of their actions? It is crucial now more than ever to reconsider our educational practices so that we may instruct new programmers to create work that is efficient, beautiful, and grounded in social frameworks, and that pays attention to materiality, in short, to do the things that so far computers have proven incapable of doing.

What computers have proven incapable of doing has been: thinking originally, deviating from the norm, contextualizing information, and recognizing beauty (Gross et al., 2009; Noble, 2018). A computer may be able to tell what is efficient, what works, and what doesn't, but it can rarely attune to what is pleasing and sensible for humans (Chandra, 2014; Coleman, 2013; Fishwick, 2008; Heiss et al., 2002). It is imperative that we teach students to think like, and unlike computers; they should be able to think critically inside and outside of computers.

As seen in Safiya Noble's *Algorithms of Oppression*, technology can structure harmful ideas, including racial hierarchies. Noble challenges the notion of algorithms as neutral entities, arguing that they can perpetuate biases due to their reliance on machine learning, which learns from human biases. She compares algorithms to language, emphasizing that they inherently carry bias and are shaped by the types of questions asked. Additionally, she highlights how neoliberal capitalist interests and advertising influence algorithms, as they are often developed by corporations that prioritize profit over ethical considerations. The opacity of algorithmic processes further complicates matters, as the inner workings are hidden from public scrutiny. Students of computer science should be aware of these issues and should be equipped with tools to mitigate them.

Computer science education cannot solely rely on traditional modes of instruction to address the aforementioned concerns; it must adapt to incorporate a multifaceted approach that encompasses project-based learning, group work, considerations of authorship, and fostering a culture of collaboration. By drawing from open-source ideologies, the art education model, and various computer science models, we must consider new and interdisciplinary ways to teach computer science. These methods should be able to respond to the three central tensions in computer science education; they should include an explanation of open-source ideologies, an understanding of ownership and remix culture, and a focus on materiality.

This paper will first examine traditional models of computer science education, as well as science education from the perspective of the arts. Then, I will lay out key tensions that arise when teaching computer science solely from the perspective of one of these models. Finally, I will propose a new model, that of "artefact" thinking, as an interdisciplinary alternative to teaching computer science.

TRADITIONS OF COMPUTER SCIENCE EDUCATION

There are three dominant traditions in computer science education; thinking of computing as engineering, as math, or as science (Sentance et al., 2023). These traditions reflect intellectual shifts in conceptualizing computing. Coming out of mathematics to support advanced calculations, it follows that computer science would relate to mathematics. With a shift towards programming as the construction of a tool, computer science is understood as a facet of engineering. When code becomes a tool to produce results, it falls into the category of science.

When we consider computer science as engineering, we aim to teach students to create things that work (Sentance et al., 2023). This can look like creating a program that performs a specific task or responds to a problem. Coding as math places more emphasis on concepts and logic. The aim is to explain to students how to make things work efficiently (Sentance et al., 2023). When computing is taught as a science, it encourages students to think about the broader context in which computer science operates. Science is often considered to be neutral and objective, seemingly above cultural and historical influences (Haraway, 1988). This suggests that scientific knowledge, including computer science, exists independently of social factors and historical context. However, this overlooks the fact that science is deeply entangled with social structures and historical contexts. The reluctance to acknowledge the social structures impacting computer science may stem from a desire to maintain the illusion of scientific objectivity and neutrality.

So what do educators currently aim to teach within computer science? The answer mostly lies in the idea of ‘computational thinking’, which can encompass logical reasoning, data literacy, and programming, depending on which lens you teach from (Gross et al., 2009;

Sentance et al., 2023). Yet all of these concepts are grounded within social and historical frameworks. Learning logic is centered around thinking correctly and rationally about problems. In computing, logic is also tied to our reliance on binary, originating from electrical engineering and the creation of digital systems. To think computationally is then bound up within applying logic, and existing in binary states. This binary thinking, while essential for many aspects of computing, can sometimes overlook the nuanced complexities of real-world problems and reinforce a rigid, black-and-white approach to problem-solving.

In assuming that science is natural, computer science becomes above reproach (Haraway, 1988). This implication of neutrality leads to an assumption that algorithms must be correct (Noble, 2018). Within the realm of candidate-screening algorithms in hiring processes, this can take the form of believing that if an algorithm selects mostly men for a position, it means that mostly men are qualified. The computer science curriculum, as traditionally followed, thus leaves very little room for questioning who we produce work for, and why.

To address some of these issues we can turn to art education, which provides alternatives for thinking about objects within social and cultural contexts (Knochel et al., 2015; Tillander, 2011). Incorporating an artistic lens has already been implemented within higher-level math education, in which professors focus on the beauty of proof rather than the ability to compute numbers; it is more important to be able to express oneself in a clear, concise, logical manner than to be able to replicate a computer's work. This framework could also prove to be beneficial in computer science education. As we enter the age of artificial intelligence, many basic tasks in computer science can be replicated through an AI agent. For instance, there are now multiple websites in which one just has to type a prompt, and an AI bot will build the web pages along with all the code needed to implement them. It is therefore now more crucial than ever to

recenter computer science towards what we, as humans, have unique abilities to do.

Integrating art practice in computing could shift much of the coding process (Graham, 2011; Gross et al., 2009; Knochel et al., 2015). Unlike traditional computing methods where projects often start with predetermined goals, the artistic process typically entails a greater emphasis on exploration and experimentation, prioritizing method over outcome (Mclean et al., 2010). However, computing education often overlooks this exploratory practice. Teaching programming through the lens of art could reintroduce and foreground this exploratory approach (Graham, 2011; Mclean et al., 2010). Embracing such an approach might introduce an architectural perspective within computer science, where exploration becomes central to making informed design decisions. This shift could lead to the development of more robust and innovative programs, as design choices are refined through iterative exploration and experimentation. By adopting the principles of artistic practice, computing education could foster creativity, curiosity, and adaptability among students, enriching their understanding and application of programming concepts.

Integrating art practice into computing could also invite a reevaluation of the place for beauty in code. Just as artists strive to create visually pleasing and aesthetically resonant works, programmers could aspire to write code that is elegant, expressive, and harmonious (Graham, 2011; Gross et al., 2009; Knochel et al., 2015). This shift towards appreciating the aesthetic qualities of code could enhance the readability and maintainability of programs. By emphasizing the importance of beauty in code, students could learn to prioritize clarity, simplicity, and coherence in their programming practices. This could help mitigate issues of technical debt, created by outdated systems too complexly intertwined and reliant on each other to easily untangle.

An artistic practice may also help reframe coding outside of success or failure. Evaluating a program's success on its ability to compile and execute does not account for whether the program is good: whether it runs efficiently, is clear, or could be read by someone new. To step outside of this framework, consider pseudocode, whose purpose is to explain a process on a high level, without getting into the nitty gritty details of implementation. While this code will never run, as it is often more in English than in any programming language, it serves as an intermediary, abstracting ideas for simplified understanding. It is certainly important to remain in touch with the functioning of the code, as even writing lines in different orders can influence the program's compilation, and affect its time complexity. But compilers are already optimized to maximize a program's performance. Understanding concepts on an abstract level, as a formulation of prose, is what remains when implementing code itself becomes the task of a computer (Gross et al., 2009).

Another benefit of teaching computer science through the lens of art education is its focus on situating knowledge in materiality (Graham, 2011; Gross et al., 2009; Tillander et al., 2011). Computer science often neglects its physical origins, and knowledge of hardware is often overlooked in curriculums (Noble, 2018). Because programming is mostly conceptualized in the digital realm, it has become increasingly difficult to situate computing in its physical space (Noble, 2018; Turkle, 2014). This is further exacerbated by terminology such as "the cloud", which creates an impression of a-physicality and lack of mass. This neglect has several implications for computing practice. For instance, there are practical reasons for which a computer scientist should understand the functioning of a computer. Understanding how memory is allocated and how processes are run can help programmers create more efficient programs.

The application of a material-centered education as seen in the arts and certain laboratory

sciences would contextualize computer science in the physical, and frame computing knowledge through a larger network of media. Teaching computer science as grounded in materiality would equally facilitate the understanding of relationships between companies and industries, as many of them are tied through channels of production. Furthermore, this could encourage students to consider environmentally aware practices when coding.

Applying an artistic or laboratory science lens can help highlight the significance of materiality. This lens is equally beneficial if we conceptualize code as a form of communication and story-telling. Yet even this framework crumbles when trying to address issues of open-source collaboration. The next section examines several issues that contemporary computer science education fails to address.

TENSIONS IN COMPUTER SCIENCE EDUCATION

There are three areas of tension in the ways that computer science is taught: the question of open-source, copying and plagiarism, and materiality. These three central areas of computer science are often ignored or misrepresented in curriculums, especially when a curriculum uses only a single paradigm. For instance, studying computer science from the perspective of art or laboratory science may make it easier for students to understand that it is important to understand the materials one uses, but this model doesn't necessarily address questions of open-source collaboration and copying. We need to combine different techniques to create a practice that can address these issues.

OPEN-SOURCE PHILOSOPHY

Central to computer science practice is the open-source philosophy, which centers on

collaborative work in which products such as source code and documentation are freely available to the public. This philosophy has significant political implications and is engrained into the very foundations of the internet. Technically, practicing computer science doesn't require knowledge of open-source principles. Yet computer science cannot exist without open-source code; it is a philosophically (and mechanically) integral element of computing. As such, students must understand its history, its ethical orientation, how to benefit from it, and how to give back.

Growing out of a hacker ethic and the culture of Bill Gates and Steve Wozniak, open source centers openness, decentralization, and the sharing of code (Eghbal, 2020). Based on ideals of the scientific process, the idea is to "free" code from proprietary control by making resources freely available to the public. This ethical standard is foundational, as open-source software is a pillar for keeping most current software running. However, open source in current practice poses certain problems. For instance, a lack of funding and acknowledgment does not motivate people to maintain code (Eghbal, 2020). Furthermore, with the rise of open-source platforms such as GitHub, the number of contributors to a project can grow exponentially, making it difficult to manage a project (Eghbal, 2020). Open source as practiced is not without contradictions or paradox: for example, while GitHub can serve as a place to share code with others and collaborate, GitHub itself runs on private (ie not open) code.

In computer science classes, open source is a rarely touched topic. Perhaps due to the fear of breaching academic integrity standards, very few classes focus on working collaboratively and teaching students how to contribute to a larger project. However, grasping open source is crucial for understanding collaborative software development. Many students miss the interconnectedness of code libraries and the ethos of free sharing in open source.

Moreover, due to open source philosophy, the concept of ownership in computer science

differs greatly from other artistic practices. Computer science, similar to the arts, blends technical proficiency and creative expression. Unlike traditional art, however, when one writes code, they don't inherently own it. One often codes for the goal of contributing to a much larger project, rather than a standalone work. In practice, coding systems are often owned by corporations, and individuals work to contribute to a pool of knowledge with limited access.

Art is not without collaborations: in fact, certain forms of art require the collaboration of the audience to create something by contributing a single line, an object, etc… yet art is typically created to be observed, appreciated, experienced, and critiqued, but the intention is rarely for it to be used or copied without permission or to be incorporated wholesale into other art. This contrasts sharply with the collaborative and shared nature of coding, where the goal is often to encourage reuse and adaptation. Only more subversive forms of art, such as remix culture, acknowledge this same level of reprocessing—and often these are ethics practiced by the reminders, not the original creators.

Introducing the concept of code as art in education prompts intriguing questions about ownership. By framing code as artistic expression, are we implicitly advocating for students to assert ownership over the code they create? Would this not contradict the entire logic of open-source code? Yet a lack of ownership in code isn’t an end-all be-all answer either; while it serves to work collaboratively and build upon others’ ideas, a lack of ownership equally diverts responsibility for mistakes. The art education model is thus not the most compatible with teaching open-source philosophy.

COPYING AND PLAGIARISM

A central difference between classroom computer science and industry practice is in the

assumption of a singular author. Schools require students to write code from start to finish without copying from outside sources (Sentance et al., 2023). This helps students think through code structure, and understand all elements that go into a project. This process also aligns with most departmental guidelines regarding academic integrity; students must not copy, cut and paste, or take from other work without citing their sources. Yet outside of academic walls (and perhaps within them too), computer scientists are well-known for their copying abilities. Coding has always been an additive, mosaic process; with Stack Overflow, GitHub, and now AI chatbots, the coding process requires the coder to first look for what has already been done, then to take and adapt it into their work. Similar to collaging, the coder must learn to select, understand, and reconfigure pieces of code to create something new.

It is not to say that code is without attribution entirely. In fact, in any project, proper convention requires a README document explaining which sections of the code do what, and what libraries are necessary to implement it[1]. Within this documentation, citations of code sources serve as further reading.

There are equally fundamental systems in place for copying from large sections of code. A primary example of this is when programmers import libraries into their projects. These libraries are collections of precompiled files and scripts that can be used to simplify coding specific tasks. For instance, importing a library to visualize graphs may be helpful in a data science project, as there are already functions in place to facilitate displaying information. Whenever a programmer imports a library, they draw upon code and functions that have already been implemented by someone else. They must then import this library into their project to use

[1] My project also contains a README file available on the GitHub repository under the title "README.md". It can also be accessed by scrolling down on the landing page of the repository. This file describes the technical aspects of the project and documents the sources used, as well as how the code is structured. It should be considered alongside this paper and the website as an important part of the project.

it, and anyone running the code must do the same. Depending on the system and library, this will have to either be a conscious, physical act, or it will automatically be done by a compiler before the code is run. Regardless of the fashion, it is highly unusual (and quite tedious) to code without using someone else's work; even a simple line, `print("Hello World")`, relies on someone else's implementation of the `print()` function to succeed. Restrictions on copying code within academic spaces are thus hard to define. Usually, the line is drawn based on where the code comes from; importing a library is usually allowed (depending on the scope of the project) while importing code from someone's GitHub project is never allowed. Yet this form of 'cheating' does not exist on the same scale when coding for a company. If the code runs, then it runs; it matters little where it came from.

The rise of Large Language Models (LLM) such as ChatGPT further complicates the issue of copying. Within academic spaces, a student cannot use a chatbot to help them write (except in specific assignments, and even then the chatbot must be credited). However, within the industry setting, it is not uncommon to use a chatbot or copilot to help you write sections of code; in fact, many companies have guidelines for their use to avoid security leaks. Most large companies have their internal chatbots, which allow employees to chat to their hearts' content without risking leaking information. This illustrates the adoption of AI chatbots in a way that is rarely seen in other industries. Emerging tools, such as GitHub Copilot further demonstrate a move towards letting the computer do the dirty work, leaving the coder to focus on the concepts they are implementing. As they advertise: code 55% faster with GitHub Copilot!

LLM copilots work similarly to the way coders search and copy code on Stack Overflow or GitHub; we find similar projects, and take what we need to produce something new. The copilot equally searches data it has been trained on and outputs a block of code that may respond

to its given prompt. The coder is then required to piece these larger blocks together and ensure that they run.

This continual cycle of borrowing, mixing, and republishing the same code is a feature existing within other disciplines as well; one could argue that most creative processes require examining other work, reacting to it, and proposing something new. The rise of LLMs and their unfettered use poses a slight threat to this system; because these programs use existing code to create new code, without strong differentiations of origin, they easily fall into endless regurgitations of error-ridden code. As a result of flawed training material, the AI can "eat its own tail" so to speak, producing gibberish, consuming that gibberish, and heaving out even more gibberish.

One issue that applies to co-pilots, and LLMs in general, is that they do not always provide reliable information. As has been documented, AI models flatten cultural differences and push ideas toward the majority (Noble, 2018). Relying on them to write code is therefore quite dangerous if you don't read over it. That being said, if you have to read over and double-check the copilot's work, what's the point of having it in the first place? Because of this, many coders will allow the copilot to write code and often neglect proofreading. Another issue then arises; when one relies on a flawed bot to write their code, the person responsible for the flawed code is still the human. We therefore have to remain cautious of using these bots as copilots for our work.

MATERIALITY

Dominant traditions of computer science ignore the politics attached to materiality—that is, they address the data but not the system that holds it (Munster, 2006; Plant, 1997; Tillander,

2011). We have become more and more removed from the computer itself as a physical object. With the rise of so-called cloud computing, the coder remains very far from the server farm running their code, making it exceedingly easy not to think about the land and resources needed. Even the terminology itself, “the cloud”, further emphasizes this abstract thinking—code is not ever stored in the air, it is always on the ground and in boxes made of material extracted from the ground. In avoiding the topic of materiality, curriculums obfuscate how things are created, and even where they go when they are discarded (Gabrys, 2013). This shift into the immaterial has not yet been accounted for in models of education (Sentance et al., 2023)

Furthermore, in ignoring the materiality of computing, our conceptualization of time and memory equally shifts (Munster, 2006). In becoming digital, time changes meaning; because information can be shared at incredible speeds, we imagine ourselves much closer to others than we are physically. Digital memory has a much larger capacity to store knowledge than our brains, or our libraries. Ever since computers have been powerful enough to store files and documents, the question has shifted from how to remember things, to how to organize what is memorized efficiently, so that information can be accessed easily (Munster, 2006). This wealth of knowledge has allowed us to share information in new ways and across space like never before. Yet it has also created an oversaturation of information; it is difficult to understand which knowledge should be focused on since there is so much to look at (Noble, 2018). Because everything can be shared digitally, it becomes up to the user, and company algorithms, to decide what is true and what is important (Noble, 2018). With this ever-enlarging memory storage system, we are rarely required to recycle information, nor question whether certain information should be discarded or updated. The truth of the matter is that the information that ends up being prioritized in these archives often re-emphasizes the most authoritative knowledge (Noble,

2018). In obfuscating histories of materiality, we forget how time and memory change when digital.

In contrast to computer science, art education is heavily focused on materiality. In fact, by introducing students to different media, art teachers aim to teach how to navigate and blend different mediums (Gross et al., 2009). The intention is that students should be able to work with a material, understand its properties, and where it comes from (Gross et al., 2009). For instance, in working with watercolor paint, teachers will introduce their students to different techniques to illustrate the properties of the paint. The student must then create their own original work by understanding the methods attached to a medium. This allows students to gain familiarity with the ingredients they work with, and learn to reuse them in new pieces. The scientific model equally emphasizes materiality. In fact, in laboratory sciences, students are expected to understand the properties of chemical substances, physical engines, or objects of the sort. In the math realm instead, similarly to computer science, concepts remain completely abstract (Sentance et al., 2023).

The application of a material-centered education could equally prove beneficial in computer science education. Because art is so heavily focused on processes and materials, integrating art education into computer science may promote a stronger attunement to hardware. This would contextualize computer science in the physical, and frame computing knowledge through a larger network of media (Sentance et al., 2023). This could equally frame computing in its human histories; exploring the histories of materials becomes integral for comprehending the evolution of computing (Gabrys, 2013). For instance, chip manufacturing was integral to the growth of Silicon Valley as a technological hotspot (Gabrys, 2013).

Teaching computer science as grounded in materiality would equally facilitate the

understanding of relationships between companies and industries, as many of them are tied through channels of production. This could encourage students to consider environmentally aware practices when computing. An extremely prominent example of computing that doesn't consider materiality is the mining of Bitcoin. These programs require immense amounts of space and energy. Perhaps if computer scientists were trained to be more aware of their consumption of physical resources, they would have more consideration for the impacts of their products.

THE ARTEFACT MODEL: AN ELEGANT SOLUTION

The solution I would like to propose to address the inadequacies of single-paradigm computer science is a version of the 'artefact model', a model which encourages students to think about computing objects within their various contexts (Sentance et al., 2023). This pedagogy aims to keep students away from thinking of computing as some magical abstraction; students must perceive computing objects within their contextual reality, in place within social and physical structures, and as specific entities rather than as general ones.

It is far too easy to spend a life staring at a screen, and never question where our creations, stored on the cloud, truly exist on the ground. Artefact thinking proposes breaking down computational objects, such as theories, concepts, and practices, and examining them not unlike art history examines artworks; there must be an understanding of how an artefact has come to be and how it stands in conversation with other artefacts. For example, when learning a new theory, students could be taught what previous theories influenced it, who came up with it and why, in what context it was established, and how it interacts with other theories. Artefact thinking requires establishing computer science as a human artefact and human creation, thus centering it within human activity.

I propose that this model should be applied as an overlay on traditional curriculums, meaning it should operate as a philosophy that informs technical curriculum choices. Through this approach, we may be able to address some of the current tensions in computer science education. When teaching coding at the introductory level, students would also learn about the history and functions of computers themselves—thinking about switch relays and binary code not as the endpoint of code, but as the origin of information. This would allow students to understand the hardware that makes their code run. This could also help students become aware of the production processes at play in producing such electronics. This focus would also imply a closer attunement to the physicality of computing processes.

Similarly, students would be introduced to open-source philosophy as a foundation of coding, and a way to understand the socio-political context of code. Looking at remix culture and the copying of code could be taught through this open-source lens, and could equally be understood as a form of familiarity with given materials, as students work code like clay to form something new.

Embracing this approach not only facilitates a deeper understanding of the subject matter but also instills critical thinking skills essential for creating effective, powerful, and aesthetically pleasing code. The artefact model offers a pathway to cultivate a new practice in computer science education—one that prepares students to create code that is not only technically proficient but also aware of its historical, social, and ethical dimensions. In combining aspects of art, science, math, and engineering education, we may develop a new practice that can prepare students to create effective, powerful, and beautiful code.

Works Cited:

Chandra, V. (2014). Geek Sublime: The beauty of code, the code of beauty. Graywolf Press.

Coleman, E. G. (2013). Coding freedom: the ethics and aesthetics of hacking. Princeton University Press.

Eghbal, N. (2020). Working in public: The making and maintenance of open source software. Stripe Press.

Fishwick, P. A. (2008). Aesthetic computing. MIT.

Gabrys, J. (2013). Digital rubbish: a natural history of electronics. Univ. of Michigan Press.

Graham, P. (2011). Hackers & Painters: Big Ideas from the Computer Age. O'Reilly.

Gross, M. D., & Do, E. Y.-L. (2009). Educating the New Makers: Cross-Disciplinary Creativity. Leonardo, 42(3), 210–215. http://www.jstor.org/stable/20532648

Haraway, D. (1988). Situated Knowledges: The Science Question in Feminism and the Privilege of Partial Perspective. Feminist Studies, 14(3), 575–599. https://doi.org/10.2307/3178066

Heiss, J. J., & Gabriel, R. (2002, December 3). The Poetry of Programming. Dreamsongs. https://dreamsongs.com/PoetryOfProgramming.html

KNOCHEL, A. D., & PATTON, R. M. (2015). If Art Education Then Critical Digital Making: Computational Thinking and Creative Code. Studies in Art Education, 57(1), 21–38. http://www.jstor.org/stable/45149256

Mclean, Alex & Wiggins, Geraint. (2010). Bricolage Programming in the Creative Arts.

Munster, A. (2006). Materializing new media: Embodiment in information aesthetics. Dartmouth College Press.

Noble, S. U. (2018). Algorithms of oppression: How search engines reinforce racism. New York University Press.

Plant, S. (1997). Zeroes + ones: Digital women + the new technoculture. Doubleday.

Sentance, S., Barendsen, E., Schulte, C., & Howard, N. R. (2023). Computer Science Education: Perspectives on teaching and learning in school. Bloomsbury Academic.

TILLANDER, M. (2011). Creativity, Technology, Art, and Pedagogical Practices. Art Education, 64(1), 40–46. http://www.jstor.org/stable/23033951

Turkle, S. (2014). Life on the screen: Identity in the age of the internet. Simon & Schuster Paperbacks.