Program or be Programmed? Teaching algorithmic principles to future designers

Christian Nold
The Open University, UK
cchristian.nold@open.ac.uk
doi.org/10.21606/drslxd.2024.059

Abstract: This paper reflects on the development of a unit about algorithmic design in the context of a new undergraduate design course at the Open University in the UK. The paper investigates how to teach algorithmic design and explores what content should be taught by carrying out a literature review of pedagogical approaches and offers reflections on their implementation in the course. The review reveals that there is currently no accepted framework for teaching algorithmic design but highlights the widespread argument for teaching a programming language. What differentiates the approaches is how one teaches programming, whether as a technical tool or as an expressive and cultural medium. The Open University unit aims to teach algorithmic principles via a student project that uses code to design wallpaper for their home. By situating computation within the student's own domestic context, the unit seeks to foster the development of personal judgment, sensitivity, and novel forms of algorithmic innovation. The paper aims to provide insights for design educators who are developing similar courses through face-to-face instruction or distance learning methods and presents a challenge to explore ways of using design pedagogy to situate algorithmic design within people's lives.

Keywords: algorithmic design; design pedagogy; computational literacy; distance learning; p5js

Introduction

In the emerging, highly programmed landscape ahead, you will either create the software or you will be the software. It’s really that simple: Program, or be programmed. Choose the former, and you gain access to the control panel of civilization. Choose the latter, and it could be the last real choice you get to make. (Rushkoff, 2010, pp. 7–8).

The writer Douglas Rushkoff argues that computer programming is a fundamental skill for future citizens to understand the world and have agency within it. This seems prescient for designers where the rise of computation and algorithmic logics have revolutionised the tools, practice, and cultural positioning of design. Yet, pedagogical approaches in design have not kept up with these changes. This short paper reflects on the development of a unit on algorithmic design, which forms part of a new undergraduate design course at the Open University (OU) in the UK that will launch in late 2024. The term algorithmic design tries to capture the new developments in arts-based programming, generative and parametric design as well as the ways that computation has changed established area such as graphic, interaction and product design.

The paper explores two questions: how to teach algorithmic design and what content should be taught. To address these inquiries, the paper conducts a literature review of existing approaches and presents preliminary reflections on their implementation in the OU course. The review reveals that there is currently no universally accepted framework for teaching this rapidly evolving field. However, it emphasises that teaching a programming language and
approaching computation as an expressive and cultural medium, are critical aspects to consider. The paper aims to provide a conceptual approach for design educators who are developing similar courses, regardless of whether they are delivered through face-to-face instruction or distance learning methods. This paper discusses solely the conceptual progress and creation of the course, with a comprehensive evaluation of this method to be presented in a subsequent paper.

**Teaching algorithmic design at a distance**

The OU is a distance learning institution that holds the distinction of being the largest university in the UK in terms of number of students. It is known for its unique approach of enabling students to study at their own pace and integrating education into their own spaces and diverse lives. This context offers specific advantages and limitations that merit consideration when teaching algorithmic design.

My background involves a decade of teaching algorithmic design as a hands-on practice to designers and architects at various universities in the UK. This teaching predominantly took place in a traditional, face-to-face studio setting, where small misunderstandings can be promptly addressed. In studio teaching, the physical presence of peers plays a key role in acquiring tacit knowledge (Mareis, 2012; Polanyi, 1962) through observing someone’s work and then attempting it oneself. On the contrary, distance learning places emphasis on the course materials having to explicitly articulate details and clarify ambiguities that are often left unspoken in a studio environment (Cross & Holden, 2020). Furthermore, incorporating successful pedagogical interventions, such as pair programming, where students directly collaborate (Vihavainen et al., 2014), becomes challenging in distance learning but not impossible (Hughes et al., 2020). These challenges of distance learning place an extra emphasis on choosing an appropriate and custom targeted pedagogical approach to algorithmic design that can teach both practice and theory.

Another noteworthy challenge is that the courses at the OU are designed to have a functional lifespan of 10 years. While some adjustments can be made in their presentation, these are limited. This means that rapidly evolving technologies like Generative Pre-trained Transformers (GPT) and courses that have emerged to respond to these technologies such as Basic Prompt Engineering with ChatGPT (Tropiano, 2023) are difficult to replicate at the OU. Instead, the OU approach to course design takes an extended view which aligns more with organisations like the Long Now Foundation (longnow.org) that use time as a deliberate reflective device. In practice this means searching for enduring principles rather than striving to keep up with the latest trends.

**Existing approaches to teaching algorithmic design**

Notably, much of the discussion on how to teach algorithmic design takes place across multiple disciplines of computer science, architecture, new media arts, media theory and software studies. What these discussions agree on is that the rise of computation and algorithmic logics creates a rupture with previous design practices and requires new pedagogical approaches. Yet, a common theme across the literature is that there are no unified frameworks on algorithmic design pedagogy (Fricker et al., 2020). Fischer and Herr (2001) for example, argue that “generative design teaching currently lacks methodologies, teaching experience and introductory study material” (2001, p. 147). One of the main dividing lines in the literature is about how to relate to teaching computer programming.

Teaching in this area historically originated from the computer sciences where an introductory course usually starts with a programming language and teaches its data structures and associated algorithms and then shifts towards higher level concepts of structuring and analysing algorithms (Baeza-Yates, 1995). While this is still a common pedagogical approach, there has been a growth in the notion of computational thinking (Wing, 2006) that does not necessarily teach hands-on programming but focuses on teaching computational principles of reduction, embedding, transformation and simulation as a universal problem-solving approach that is applicable to students across all disciplines (Buitrago Flórez et al., 2017). The notion is that all problems can be translated into computational challenges and a lot of focus is placed on trying to embed computational terms into everybody’s vocabulary: “when your daughter goes to school in the morning, she puts in her backpack the things she needs for the day; that’s prefetching and caching” (Wing, 2006, p. 34). This approach to computation, as a mode of thinking, has been popular with school educators across the world (Bower et al., 2017; Grover & Pea, 2013).

Yet, Brennan and Resnick (2012) argue that there is little agreement about what the practices of computational thinking actually consist of. Based on their experience of developing the Scratch programming environment (2023) they suggest three elements that are more grounded:
**Computational concepts** (the concepts designers engage with as they program, such as iteration, parallelism, etc.), **computational practices** (the practices designers develop as they engage with the concepts, such as debugging projects or remixing others’ work), and **computational perspectives** (the perspectives designers form about the world around them and about themselves) (Brennan & Resnick, 2012, p. 1 emphasis added).

In this approach, computational literacy involves hands-on programming as well as focusing on students to develop a reflective judgment on their actions and impacts on the world. Michael Mateas (2005) extends this grounded approach by proposing a notion of procedural literacy as “the ability to read and write processes, to engage procedural representation and aesthetics, to understand the interplay between the culturally-embedded practices of human meaning-making and technically-mediated processes” (2005, p. 67). From this media arts perspective, computation is an expressive medium with its own history and affordances. Mateas argues that procedural literacy needs to be taught through hands-on programming as well as discussions on media theory texts about the cultural and historical context of computation. It is only through this combination of practice and theory that students learn to navigate the huge tower of abstraction that exists in any computer system, with each layer defining its own little process universe, and with all layers, including the programming languages themselves, contingent human-authored artifacts, each carrying the meanings, assumptions, and biases of their authors, each offering a particular set of affordances (Mateas, 2005, p. 70).

Mateas argues that this approach develops a technical and cultural sensitivity at the same time which builds the student confidence to engage with the complexity of computer systems. This culturally situated approach to computing is built on the foundation of software studies (Fuller, 2008) and media theory (Manovich, 2002). Lev Manovich’s contribution has been highly influential in providing a cultural history of new media and proposing it as a language characterised by five principles: numerical presentation, modularity, automation, variability and transcoding (2002). Manovich describes automation as a principle where “human intentionally can be removed from the creative process, at least in part” (2002, p. 53) and extends from low-level batch processing images in Photoshop, to high-level automation via AI agents. The importance of automation is highlighted by many authors who recognise the displacement of the hands-on designer into a creative system (Fischer & Herr, 2001). Yet the distinctiveness of the media theory approach, is that it articulates computing as culturally and historically constructed and related to semiotics and Henry Ford’s factory logic (Looy, 2003). However, trying to integrate theory and practice can pose challenges; Mateas admits that “it can be difficult for students to think about the historical origins of the graphical user interface and its relationship to cybernetic discourse while connecting this back to the nitty-gritty details of writing code” (2005, p. 80).

One of the most significant pedagogical approaches is the use of hardware and programming languages that are specifically designed for educational purposes rather than production. Logo was developed in 1967 (Papert, 1980) as an attempt to embed Piaget’s constructivist educational philosophy (Lourengo, 2014) into a pedagogical object-to-think-with (Beynon, 2017). The contemporary successor to this approach is the Processing (2023) programming language. Processing has evolved into a global movement with its own foundation, extensive support forums, associated design courses, and spinoff programming languages. The development of Processing originated from John Maeda’s aesthetic approach to programming that emphasises playful visual experimentation (Maeda, 2001, 2004). Montfort and colleagues argue that Processing serves as a bridge, allowing individuals with programming backgrounds to learn about visual imagery, while enabling artists to gain procedural literacy (Montfort et al., 2009, p. 106). The Processing language is a specialised pedagogical device that enacts these goals in the way it makes it quick and easy to draw to the screen. Its philosophy is that “you can’t learn to code just by reading about it—you need to do it” (McCarthy et al., 2016, p. 9). This approach is replicated in the associated curricula (Pereira, 2017; Shiffman, 2023; UCLA Arts, 2023) and books (Reas & Fry, 2007; Shiffman, 2008), which prioritise the practical activity of drawing visual shapes on the screen before progressing to more abstract topics such as programming structures of loops and arrays. The book Algorithms for Visual Design (Terzidis, 2009) is one of the exceptions, taking a more conceptual and structural approach to teaching Processing.

Despite Processing’s valuable pedagogical attributes, it is important to note that most teaching resources focus on mastering the programming language itself while side-lining the social contexts of computation, or telling a narrow history of computational image-making that starts in the 1960s (McCarthy et al., 2016, p. 3). Thus, while Processing is a useful pedagogical tool that can be employed within various approaches, it does not inherently integrate practice and theory but requires explicit pedagogical materials to create these links.
Developing the OU course

The OU course on algorithmic design uses insights from the literature to frame computing as both an expressive and cultural medium and involves students in hands-on programming. It uses programming as an object to both think-with and to play-with rather than aiming to turn the students into computer scientists. The course material provides the students with functioning code examples and asks them to creatively tweak (Lau et al., 2021) and tinker (Gutwill et al., 2015; Lyon et al., 2018) with the code. This means the students never have to write code from scratch. Instead, the focus is on encouraging the students to play and experiment with tweaking the code while avoiding technical programming terms and complex maths. The course focuses on the students exploring the creative potential of algorithms, demonstrate their judgement about using serendipity in the design process, and positioning their work in the social and historical context of computation. To support this contextual approach, the course presents the students with a set of six high level principles: Automation, Repetition, Variability, Modularity, Serendipity, and Emergence. These terms build on Manovich’s principles (Manovich, 2002) as well as Mateas’ notion of procedural literacy (2005). In the OU course, these principles are illustrated by historical examples as well as interactive ‘widgets’ that allow the students to experience these principles.

The main challenge faced during the design of the course was how to relate these abstract principles to the lives of the diverse OU students with busy and complex lives, who often manage multiple jobs and caregiving responsibilities. To address this challenge, the core of the OU course is a practical project that asks the students to use algorithms to design wallpaper for their own home. Using the Processing-derived p5js (2023) language, the students learn to program their own wallpaper within their browser-based Virtual Learning Environment. Once they have created a design, they can print on A4 sheets with a home printer and assemble the sheets to create the wallpaper pattern. As a contextual and practical starting point, the students are provided with information about the tile designs created by Sébastien Truchet, an 18th century monk polymath, as well as p5.js computer code that can be used to recreate these designs, see figure 1.

Figure 1. Screenshot from the preliminary OU teaching material. The students use the same encoding system as Sébastien Truchet to recreate the historical tile patterns using p5.js code. The programming editor is embedded inside the Virtual Learning Environment to keep the coding and historical context together.
The provided code functions as a springboard for the students to create their own wallpaper designs by tweaking this code using their own tile designs. The idea is that by focusing on algorithms in the context of the student’s own domestic space, that this encourages personal judgment and triggers discussions with family members about the aesthetics and relevance of these patterns, see Figure 2. The collective sharing of these wallpaper designs by the students can also be used by the design tutors to open discussions about the cultural assumptions about computation and the forgotten non-western history of tiling and fractal designs (Eglash, 1999). The goal is for the OU course to create a pedagogical environment where algorithmic logics become reshaped through the situated contexts of the students’ lives that allow them to create new forms of innovation that spring from these situations.

Figure 2. Screenshot from the preliminary OU course material showing algorithmic wallpaper that has been temporarily attached to the inside of a kitchen cabinet. As part of the assessment the students are asked to reflect on how their pattern designs relate to the location where they are placed in the home.

Discussion

While the writer, Douglas Rushkoff, presents us with a binary choice between programming or being programmed, the answer seems to be not so straightforward. Instead, what actually matters is how one learns to program, whether as a technical tool or as an expressive and cultural medium. The literature agrees that teaching hands-on programming is crucial as it aids in developing computational literacy. Yet there is disagreement around how to integrate the cultural context and history of computation, personal judgment, and reflexivity, alongside practical programming. The approach demonstrated in the OU course tries to entangle computation within the domestic context. It is thus a kind of situated design (Nold, 2023) that builds on ideas of domestic computing (Gaver et al., 2008) to remake computing through the lens of everyday life. This approach to teaching algorithmic design tries to offer contextual relevance to a diverse range of students and presents a practical alternative to computational thinking approaches that seek to push computational jargon into daily life.
Acknowledgements: I would like to thank the two anonymous reviewers for commenting on an earlier version of the paper. The paper has also benefited from discussions with Derek Jones and Iestyn Jowers, both from the Open University.

References


Program or be Programmed? Teaching algorithmic principles to future designers

Intentionally Left Blank, 12-14 June, Linköping University, Norrköping, Sweden.
https://doi.org/10.21606/nordes.2023.76


About the Author

Christian Nold is a researcher, artist and designer inventing models and technologies for collective representation. He is a Lecturer in Design at the Open University and known for the Bio Mapping project which was staged with thousands across sixteen countries.