Deconstruction/Reconstruction: A Pedagogic Method for Teaching Programming to Graphic Designers

Stig Møller Hansen, Ph.D. Student
Department of Digital Design and Information Studies, Aarhus University, Denmark
www.au.dk
e-mail: smh@cc.au.dk

Abstract

This paper proposes, describes and exemplifies a hands-on, experiential pedagogic method, *deconstruction/reconstruction*, specifically designed to introduce graphic design students to programming in a visual context. The method uses pre-existing commercially applied graphic design specimens as its main material to contextualize programming into a domain familiar to the audience. Observations of the method used in teaching are discussed, and its potential evaluated based on feedback provided by the students.

1. Introduction

Being code-literate is considered a crucial ability in today's society. Permeating through all parts of contemporary culture, this view is also influencing the education of graphic designers, prompting students to recast their existing skills to fit the medium of the code and educators to develop new courses that help build this literacy [1, 2, 3]. However, most graphic design students perceive programming as an abstruse skill they will never be able to master, and have a hard time trying to connect the activity of programming with the essence of their profession; crafting visual artifacts. Although many attempts have been made to teach programming to a visually oriented audience, most of them use seemingly random layouts, bouncing balls or simple characters in monochrome color schemes (e.g. [4, 5, 6]) to illustrate programmatic principles. To an audience, who equate a lack of aesthetics with a lack of relevance, neglecting the importance of the visual quality causes them to lose interest. To encourage graphic designers to explore programming as a creative tool, it is vital that new teaching strategies be developed, tailored to fit how this specific audience acquires new knowledge. In a contribution towards building computational literacy among graphic designers, this paper proposes and describes a hands-on experiential pedagogic method, deconstruction/reconstruction, specifically designed to introduce programming in a visual context.

2. Background and influences

For nine years I have taught introductory programming classes to undergraduate graphic designers at The Danish School of Media and Journalism. During this time, I have observed some recurring critical issues that negatively affect student retention, engagement, and learning outcome:

- Students find it hard to relate the activity of programming to their line of work.
- Students feel intimidated by the prospect of working with mathematics, logic, and structure.
- Students respond poorly to a lack of aesthetic quality in the output produced by their code.
- Students are easily distracted when asked to consider aesthetic issues. They quickly obsess over design-related issues, forgetting that their primary goal is to learn how to program.
- Students lack a starting point for their knowledge construction. As novice programmers they spend their time in the bottom half of Anderson and Krathwohl's Taxonomy [7], not yet in a position where they feel confident about programming to be creative with it.
- Students respond negatively to passive auditorium lectures and abstract, verbal explanations.
- Students are deterred by strange syntax and indecipherable error messages.

Seeking to alleviate these issues, I decided to develop a new pedagogic method specifically tailored to accommodate the learning needs of my students. To inform the design of the method, I summarized my observations into a set of guidelines:

- The link between programming and crafting of visual artifacts must be clearly visible.
- The output of the programming exercises must be visual
- The output must possess an aesthetic quality that makes it useful and sellable at a professional level.
- Students must be given an "object-to-think-with" [8], a cognitive artifact to serve as a link between their pre-existing internalized mental structure ("how to create graphic design") and the formation of new abstract knowledge ("how to program").
- Students must be given a fixed goal to provide a clear focus. Also, a fixed goal can serve as a measuring stick allowing students to continuously evaluate their progress.
- Students should not be asked to consider aesthetic issues to keep them focused on learning how to program.
- Mathematics, logic, and structure should only be taught when the students encounter a need for it, preferably by letting the students investigate the topic themselves, guided by the teacher.
- Students must be given the same material to encourage sharing of knowledge and discussion around a common base.
- Students must be actively engaged in the task of programming to build hands-on experience.
- Students must work in a programming environment that provides a low threshold (easy entry to usage for novices), high ceiling (powerful facilities for sophisticated users), and wide wall (a small, well-chosen set of features that support a wide range of possibilities) [9].

I chose to build the method around the recreation of pre-existing design specimens. This decision resolved several issues at once: It established a direct link between programming and design, introduced a relatable "object-to-think-with" that doubled as a fixed target, thus eliminating the risk of students losing focus by being having to make aesthetic choices.

Constructionism was chosen as the theoretical foundation of the method. Among other things, constructionism let students use the information they already know ("how to create graphic design") as a foundation for acquiring more knowledge ("how to program") in a different domain. Also, constructionism holds that learning happens most effectively when students are active in making external artifacts they can reflect upon and share with others. Finally, constructionism prescribes that the teacher must take on a mediational role as opposed to an instructional role, assisting students to individually understand problems in a hands-on way.

Guzdial [10, 11] suggest that teaching programming needs to be contextualized and meet the needs of the learners. The target audience is intended to merely be “programming tourists,” [12], thus a rigorous adherence to “correct” Computer Science terms was abandoned in favor of a terminology that better helped students build cognitive models of programmatic principles. Another key factor in favor of contextualization is to make apparent the usefulness of programming in the student's profession.

A term introduced by Papert [8] and later popularized by Wing [13], Computational Thinking deals with thought processes involved in formulating a problem and expressing its solution(s) in such a way that a computer—human or machine—can effectively carry out [14]. Key principles in Computational Thinking are:

- Decomposition (breaking down a complex problem into smaller, more manageable parts)
- Pattern recognition (looking for similarities among and within problems)
- Abstraction (focusing on the important information only, ignoring irrelevant detail)
- Algorithms (developing a step-by-step solution to the problem, or the rules to follow to solve the problem).

These principles influenced the design of the method and are embedded in the activities therein.

Finally, the work of Stahl [15] also informed the design of the method. According to Stahl, transforming tacit preunderstanding into a computer model happens in a series of successive steps. In his discussion, Stahl, among other things, suggests a taxonomy of classes of information [15, pp. 178-183]. This taxonomy greatly inspired the design of the method to be a number of sequential steps divided into two distinct phases.

3. Method described

The deconstruction/reconstruction method consists of two successive phases, deconstruction, and subsequent reconstruction. Each phase has three steps. Activities associated with each step are briefly described in figure 1. A detailed account of how the method is applied in practice is given in section 4 of this paper.

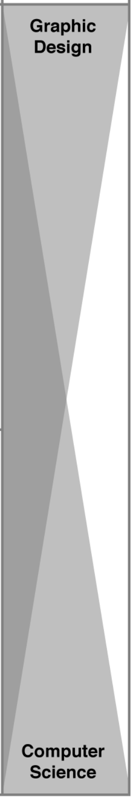
Phase	Step	Activity	Material	Domain
Deconstruction	1	SELECT Choose a pre-existing graphic design product specimen to be deconstructed from the sample set provided by the course instructor.	Paper and pen	
	2	DESCRIBE Make detailed notes about immediately visible visual components, e.g., shapes, typography, colors, scaling, rotation, grids, rhythm, and repetitions.		
	3	ANALYZE Identify and formalize invisible components, e.g., interconnections, math, logic and rules required to control the design system and make it behave as desired.		
Reconstruction	4	CONVERT Convert notes from steps 2 and 3 into code that replicate the chosen specimen. Use the original specimen as a visual reference to guide and evaluate the process.	Code	
	5	EXPLORE Modify the variables in the design system to create alternate versions of the original specimen.		
	6	TINKER Modify (and possibly break) the code to create radical mutations of the original specimen.		

Figure 1: Schematic overview of the deconstruction/reconstruction method.

The purpose of the deconstruction phase is to keep the students in their comfort zone by letting them rely on their pre-existing knowledge of graphic design principles and terminology to deconstruct an existing design product to form the basis of the reconstruction phase. The purpose of the reconstruction phase is to let students discover programming as a practical craft acquired by incremental conversion of their notes from the deconstruction phase into code, thereby constructing a self-contained design system capable of reproducing the chosen specimen, and acting as a platform for playful discovery through manipulation of variables and the code itself.

As the student completes each step, he/she gradually shifts from using their existing skills in a familiar domain (Graphic Design) toward acquiring new skills in an unknown and unfamiliar domain (Computer Science).

Material

As its main material, the method uses pre-existing commercially applied graphic design specimens. Examples of these are posters, packaging, logos, typography, signage, bank notes, stamps, etc. Specimens are handpicked by the teacher based on their ability to be deconstructed, meaning that they must exhibit distinct visual characteristics indicating that an underlying system or set of rules has played a key role in their creation. Specimens should be easily replicable using geometric primitives, basic linear transformations (e.g., translation, rotation, scaling) and control flow statements (e.g., decision-making, looping, branching). A selection of suitable specimens that meet these criteria is shown in figure 2 to provide an idea of the visual genre.

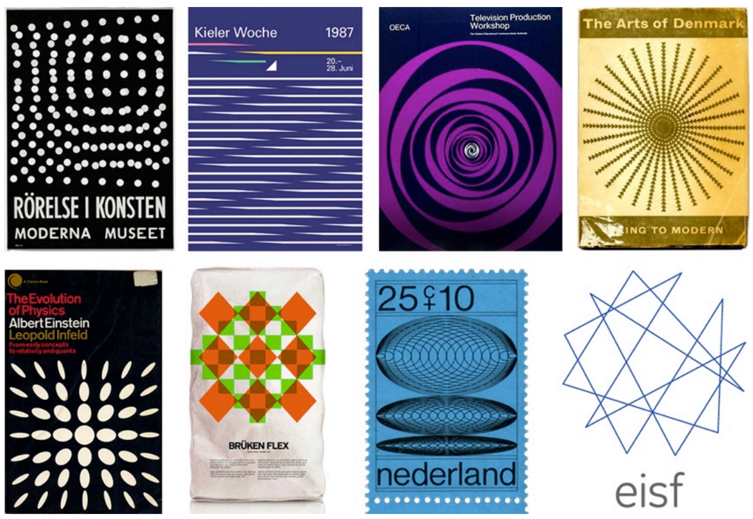


Figure 2: A selection of specimens suitable as material for the method.

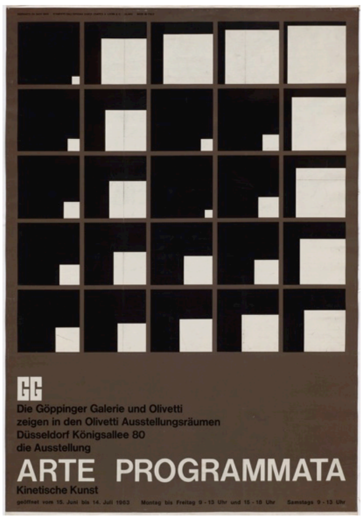

GG
Die Göppinger Galerie und Olivetti
zeigen in den Olivetti Ausstellungsräumen
Düsseldorf Königsallee 80
die Ausstellung
ARTE PROGRAMMATA
Kinetische Kunst

Figure 3: Poster by Enzo Mari (1963)

4. Method exemplified

In this section, the activities associated with each step of the deconstruction/ reconstruction method are discussed using Enzo Mari's 1963 poster "Arte Programmata: Kinetische Kunst" [16] (figure 3) as example. Processing [17], a popular Java-based language for learning how to code within the context of the visual arts, is used as the programming environment.

Step 1: Select
Guided by his subjective aesthetic preference, a student, Peter, chooses the Arte Programmata poster from the set of specimens provided by the teacher.

Step 2: Describe
Taking notes using pen and paper, Peter describes the poster's immediately visible components:

- "The poster is portrait format."
- "The background color is brown."
- "The upper part of the poster contains one 5x5 grid of black squares with inset spacing taking up the entire width of the poster excluding a border margin."
- "Each black square contains one white square of varying size."
- "The white squares increase then decrease in size while forming a spiral pattern."
- "The white square is fixed to the lower right corner of the black square."
- "The lower part of the poster has a white all-caps title spanning the entire width of the poster excluding the border margin + an additional black text set in a small font size aligned to the left."
- "Separating the 5x5 grid and the typography is a small white logo aligned to the left."

Peters observations are described using graphic design terminology familiar to him. Embedded in his description are clues about features that he must consider in his code (e.g. "square," "grid," "border margin," "inset spacing".)

Step 3: Analyze
Still using pen and paper as his material, Peter identifies and formalizes the underlying math, logic and rules needed to construct the poster. In the previous step, Peter loosely described a spiral pattern of oscillating white squares. In this step, he must make additional considerations to explicitly describe this spiral pattern: Is it rotating left or right? Does it go inside out or outside in? Where are its starting and ending points? Also, looking at the oscillating squares: How many oscillations? What are the minimum and maximum size? What principle is used to calculate the rate of change in size: Sine waves? Linear interpolation? Exponential change? These observations do not translate into simple built-in commands. They require rules to be established and algorithms developed. To formalize a thing like oscillation, something that is otherwise easily (but imprecisely) verbalized, Peter is forced to look into mathematics of oscillating functions, realizing that even a seemingly simple thing like oscillating movement can be accomplished using many different techniques all of which ultimately affect the visual style of the output. No code is written yet, although, during his research, Peter comes across a pseudocode spiral algorithm that helps him understand how spiral patterns are constructed in a two-dimensional grid.

Step 4: Convert
In this step, Peter launches Processing, as he transitions from paper and pen to code. By using his notes from previous steps as starting point, Peter gets an idea of what his program must contain and do. Sampling the original artwork, he converts colors from broad descriptions to specific color codes ("Brown" = `#5A4531`, "White" = `#F7F1E5` and "Black" = `#000000`). Squares are drawn using the built-in `rect()` command. The 5x5 grid is constructed using two nested `for()`-loops representing x-coordinates and y-coordinates respectively. To correctly place the black and white squares, functions like `pushMatrix()` and `popMatrix()` in conjunction with `translate()` is used. Investigating the `sin()`-function, Peter chooses a sine wave moving from 0 to π to achieve the oscillating white squares. In search of a way to mimic the spiral pattern, Peter modifies pseudocode found online to fit his needs. The typography can be made either as text or inserted as an image. Painstakingly recreating complex typography letter by letter serves no point; also, students might get distracted from programming when trying to correctly identify, download and install the font. Therefore, in this example, Peter was asked to simply cut out the original typography as a separate image using Photoshop, and insert it into his program as a static image. As Peter converts his notes from steps 2 and 3, he gradually constructs a program capable of recreating the original specimen. Besides acting as an "object-to-think-with," the original poster also doubles as a visual reference used by Peter to measure his progress and evaluate the behavior of his program.

Step 5: Explore
In this step, Peter must produce alternative versions of the original poster without modifying his code. By only changing variables, in this particular case using Processings "Tweak Mode," instant feedback is provided allowing for real-time exploration of the solution space inherently described by the code. A set of Peter’s

possible alternatives to the original specimen, obtained by tweaking the variables in his code, can be seen in figure 4.

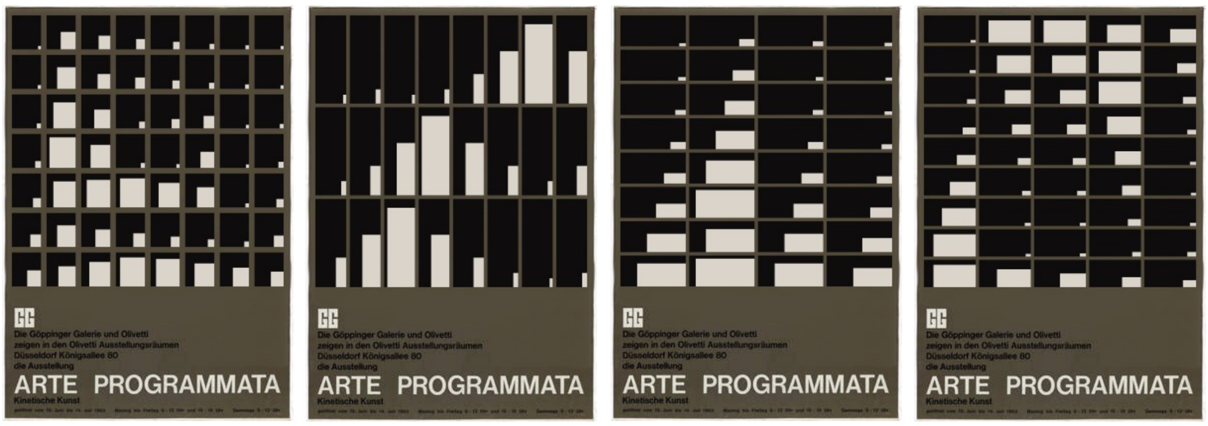

GG
Die Göppinger Galerie und Olivetti
zeigen in den Olivetti Ausstellungsräumen
Düsseldorf Königsallee 80
die Ausstellung
ARTE PROGRAMMATA
Kinetische Kunst

Figure 4: Alternative versions obtained by tweaking variables.

Step 6: Tinker
Having gained an understanding of the "mechanics" of the code, Peter begins modifying the code itself. Now, more radical solutions emerge. The result of Peters' tinkering with his code as well as continued tweaking of the variables can be seen in figure 5.

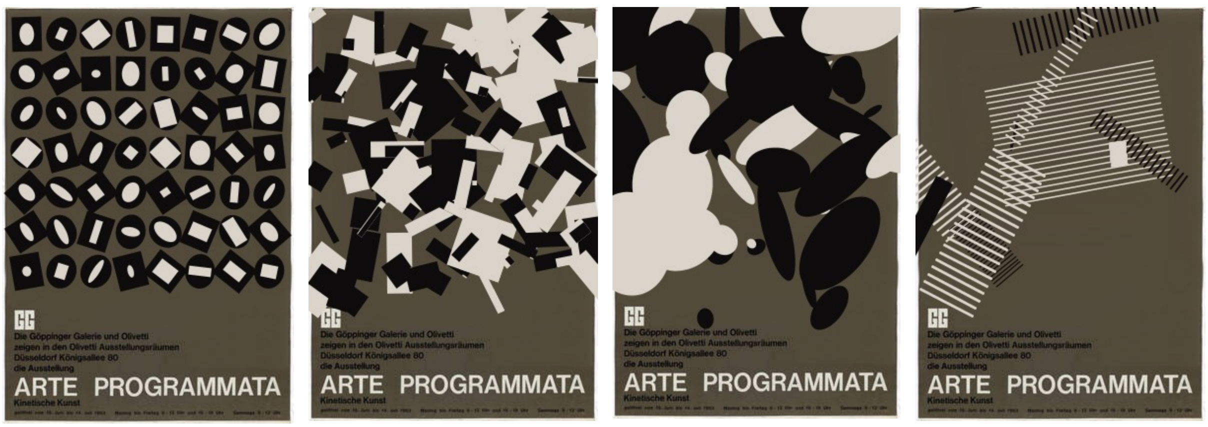

GG
Die Göppinger Galerie und Olivetti
zeigen in den Olivetti Ausstellungsräumen
Düsseldorf Königsallee 80
die Ausstellung
ARTE PROGRAMMATA
Kinetische Kunst

Figure 5: Alternative versions obtained by modifying code and tweaking variables.

5. Method used in teaching

I used deconstruction/reconstruction method in two introductory programming courses taught at The Danish School of Media and Journalism. Participants were classes of 20-24 undergraduate graphic design students (ages ranging between 21-33 years, 50/50 gender ratio) with little to no prior programming experience. The aim of the courses was to equip the students with sufficient cognitive and practical skills to enable them to conceive and execute custom made code-driven design systems. The deconstruction/reconstruction method was used as a recurring daily exercise in the first week.

As prescribed in the method, I chose a sample set of 20 pre-existing graphic design specimens from a curated collection [18]. The entire set of specimens made available as handouts and digital files to the students is shown in figure 6.

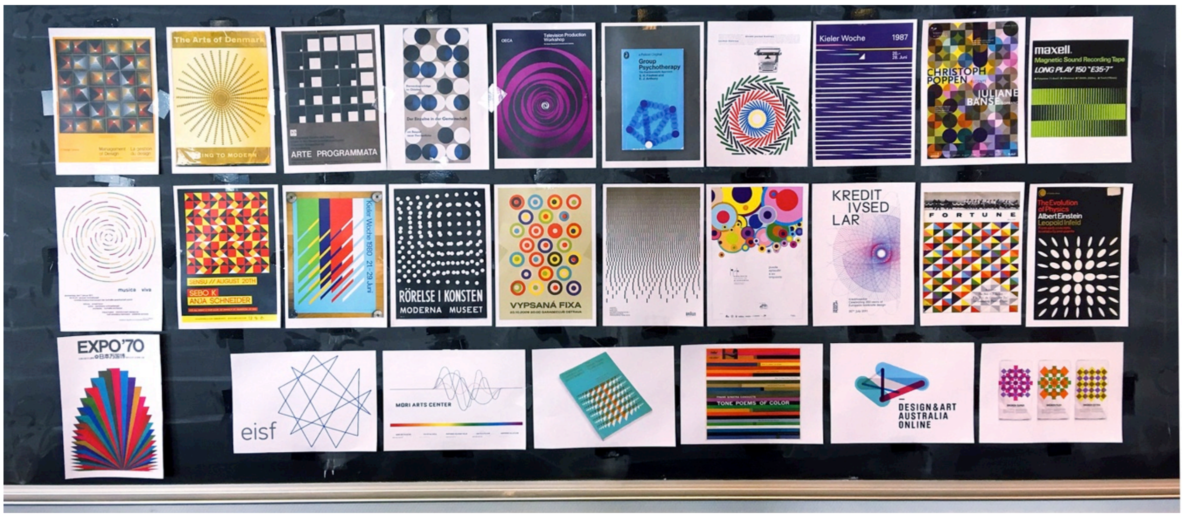


Figure 6: The collection of chosen specimens taped to the blackboard in the studio provided a quick visual overview.

Step 1: Select
Initially, choosing a specimen was a simple matter of personal preference and daily mood. Later, the students' choice was influenced by their newly acquired skills. If they had learned how to make a two-dimensional grid, students tended to choose a specimen that would allow them to reuse this programmatic feature in addition to posing a new challenge.

Step 2: Describe
The students felt confident as they began to describe their chosen specimen. Trained observers of graphic design, students had few problems describing the immediately visible components. Perhaps overly confident in their own ability to memorize their findings, I found it necessary to stress the importance of noting all observations on paper. Students spontaneously developed the habit of using Photoshop's eraser and cloning tool to remove all design components besides the background and typographic elements. This provided an authentic background to import in step 4 to make the output look almost identical to the original specimen.

Step 3: Analyze
Students began leaving their comfort zone when asked to explicitly describe the math, logic, and rules of their chosen specimen. Certain relations and behaviors were easily described using basic mathematical principles (e.g., sine/cosine, Pythagoras, linear transformations) while others relied on formulas or phenomenon one could not expect the students to know beforehand (e.g., Fibonacci series, recursion, moiré). I assisted the students in researching any formulas or techniques they might need to recreate the specimen, being careful not to provide explicit answers. This step provided a great opportunity to for the students to practice and utilize Computational Thinking principles as discussed in section 2 of this paper.

Step 4: Convert
Launching Processing and converting notes into code, students gradually discovered how variables, arrays, functions, classes, as well as other programmatic building blocks, helped them extend their static system to become a fully functioning, dynamic system capable of replicating the original specimen. This step was – without a doubt – the most challenging step for the students. They spent the majority of the time working on the daily assignment completing this step, slowly grasping programming logic, structure, looking up syntax in the language reference, and tracking down bugs.

Step 5: Explore
In this step, students used Processing's 'Tweak Mode' to manipulate variables with instant visual feedback. They would bend, stretch and inevitably break their programs. Immersing themselves in playful experimentation, students kept generating new variations from the seemingly infinite number of possibilities, always curious to discover what output their system would generate next. Students were asked to capture a visual log of their progress to show the extent of the visual diversity that their system was capable of producing. Examples from a students' visual log are shown in figure 7.

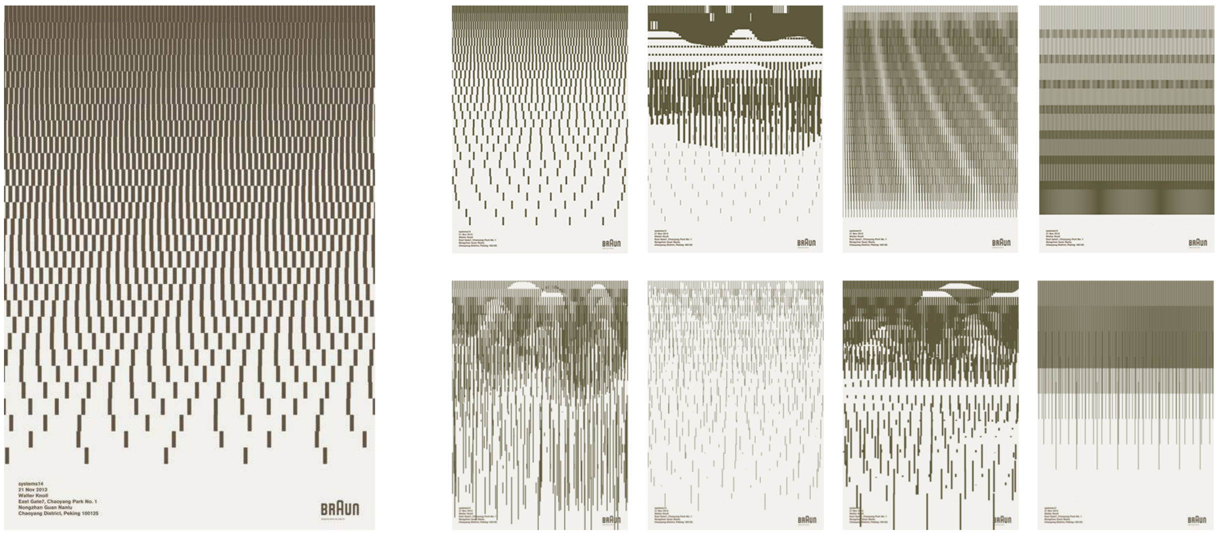

systems14
21 Nov 2012
Walter Knoll
East Gate7, Chaoyang Park No. 1
Nongzhan Guan Nanlu
Chaoyang District, Peking 100125
BRAUN

Figure 7: A students attempt at recreating the original specimen (big image, left) [19] using code, and his subsequent experiments modifying the identified variables and the code itself to produce radically different versions (small images).

Step 6: Tinker
Spurred on by their active experimentation in step 5, students began to modify the code itself. Through this process, students discovered that code, although immaterial and intangible, still possess plasticity and is highly malleable. Their confidence in their abilities grew, and this kind of tinkering and hacking was encouraged to support their urge to experiment. This step gave occasion to discuss topics like version control, optimization and advanced debugging.

Most students managed to work through steps 1-6 in one day (= 7 hours of scheduled and supervised studio time). On a few occasions, students gave up trying to complete the daily assignment. This was mainly due to issues arising in step 4 as a result of their lack of experience.

True to constructionist learning theory, students were asked to share their experiences with fellow students, currently trying to solve the same specimen. This had them verbalize and explain how they had arrived at a solution, further anchoring their understanding of what they did.

6. Concluding remarks

In this paper, a pedagogic method for teaching graphic designers programming in a visual context has been outlined and put into practice. Supported by an overall positive student response expressed in follow-up plenary interviews, the method appears as a promising way of introducing graphic design students to programming in a visual context.

The idea of contextualizing programming using pre-existing graphic design specimens was well received. Students entered their programming course with skepticism and anxiety, but introducing the deconstruction/ reconstruction method and explaining how it relied on familiar and well-known material defused the student's immediate aversion to code. The students also appreciated being given a real-life case as a starting point and step-by-step method to guide their learning process.

Though praised by the students, it can be argued, that repetitiously remaking work done by other graphic designers does not stimulate them to synthesize their knowledge into new independent creations. While this might be true, the deconstruction/reconstruction method is primarily designed to keep students engaged and motivated while introducing them to the nuts and bolts of programming. If students, by the rote learning and repetitive practice implicitly inscribed in the method, manage to cognitively link visual patterns with basic programmatic techniques, they have established a solid basis for taking full advantage of the creative potential of computational media in their future line of work.

To further put the social and learning-through-sharing ideas of constructive learning theory in play, one possible future improvement would be to make the deconstruction phase group-based to incite discussion and make problem-solving a more verbal exercise. Moving to the reconstruction phase, shifting to individual work will still allow for a personal hands-on experience with programming. Having multiple students working individually in parallel to implement a jointly deconstructed specimen will further increase the chances of students helping and learning from each other.

7. References

1. Tober, B. (2012): *Making the Case for Code: Integrating Code-Based Technologies into Undergraduate Design Curricula*. Abstracts & Proceedings from the Eigth Annual UCDA Design Education Summit.
2. Pettiway, K. (2012): *The New Media Programme: Computational thinking in Graphic Design Practice and Pedagogy*. Journal of the New Media Caucus, CAA Conference Edition 2012.

3. Freyermuth, S. S. (2016): *Coding As Craft: Evolving Standards in Graphic Design Teaching and Practice*. Plot(s), Volume 3, 2016, pp. 57-71. Parsons School of Design, New York, USA.
4. Reas, C. & Fry, B. (2014): *Processing: A Programming Handbook for Visual Designers, Second Edition.* MIT Press, Cambridge, Massachusetts, USA.
5. Shiffman, D. (2015): *Learning Processing, Second Edition: A Beginner's Guide to Programming Images, Animation, and Interaction*. Morgan Kaufmann, Burlington, Massachusetts, USA.
6. Reas, C. & Fry, B. (2015): *Make: Getting Started with Processing, Second Edition*. Maker Media, San Francisco, California, USA.
7. Anderson, L. W., & Krathwohl, D. R. (2001): *A taxonomy for learning, teaching, and assessing: A revision of Bloom's taxonomy of educational objectives.* New York: Longman.
8. Papert, S. (1980): *Mindstorms: Children, computers, and powerful ideas*. Basic Books, Inc.
9. Resnick, M. et. al. (2005): *"Design Principles for Tools to Support Creative Thinking"* in "Creativity Support Tools - A workshop sponsored by the National Science Foundation June 13-14, 2005, Washington, DC."
10. Guzdial, M. (2007): *Contextual computing education increasing retention by making computing relevant*. White paper, Georgia Institute of Technology.
11. Guzdial, M. (2010): *Does contextualized computing education help?* ACM Inroads, 1(4), 4-6.
12. Amiri, F. (2011): *Programming as design: The role of programming in interactive media curriculum in art and design*. International Journal of Art and Design Education, 30(2), 200-210.
13. Wing, J. (2006): *Computational thinking*. Communications of the ACM, 49(3), 33-35.
14. Wing, J. (2014): *Computational Thinking Benefits Society*. 40th Anniversary Blog of Social Issues in Computing. (Retrieved November 4, 2017, from http://socialissues.cs.toronto.edu/index.html%3Fp=279.html)
15. Stahl, G. (1993): *Interpretation In Design: The Problem of Tacit and Explicit Understanding in Computer Support of Cooperative Design.* PhD dissertation in Computer Science, University of Colorado, August 1993.
16. Mari, E. *Arte Programmata, Kinetische Kunst*. (1963) Printed by Officina d'Arte Grafica A. Lucini e C, Milan (Retrieved November 3, 2017, from https://www.moma.org/collection/works/8052?locale=en)
17. Processing. (Retrieved November 3, 2017, from: http://www.processing.org/)
18. Pinterest. *Computational Graphic Design Inspiration*. (Retrieved November 3, 2017, from: https://www.pinterest.dk/stixan/computational-graphic-design-inspiration/)
19. Lee, J (2014). *systems 14*. (Retrieved November 7, 2017, from http://www.leejaemin.net/systems-14)