MADE:

Design Education & the Art of Making

26th National Conference on the Beginning Design Student

College of Arts + Architecture
The University of North Carolina at Charlotte

18–21 March 2010

PROCEEDINGS
2010

26

MADE:
Design Education & the Art of Making

MADE: Design Education & the Art of Making examined the role of making past, present & future, both in teaching design and in the design of teaching. The conference addressed theories & practices addressing fabrication & craft in all studio disciplines, and to take measure of their value in pedagogies of beginning design.

Paper presentations delivered a set of eight themes derived from the overall focus on Making. The team of moderators drove the agenda for these themes, and arranged paper presentations into specific sessions indicated by the schedule. Abstracts were reviewed in a blind peer-review process.

Conference co-chairs:

Jeffrey Balmer & Chris Beorkrem

Keynote speakers:

Simon Unwin
David Leatherbarrow

Offered through the Research Office for Novice Design Education, LSU, College of Art and Design, School of Architecture

Session Topics

Making Real
Moderator: Greg Snyder
Making Virtual
Moderators: Nick Ault, David Hill
Making Writing
Moderators: Nora Wendl, Anne Sobiech-Munson
Making Drawings
Moderators: Thomas Forget, Kristi Dykema
Making Pedagogy
Moderator: Michael Swisher
Making Connections
Moderator: Janet Williams, Patrick Lucas
Making Masters
Moderators: José Gamez, Peter Wong
Making the Survey
Moderators: Emily Makas, Rachel Rossner
Open Session
Moderators: Jennifer Shields, Bryan Shields

Paper abstract reviewers

· Silvia Ajemian · Nicholas Ault · Jonathan Bell · Julia Bernert
· Gail Peter Borden · Stoel Burrowes · Kristi Dykema
· Thomas Forget · Jose Gamez · Laura Garafalo
· Mohammad Gharipour · David Hill · Tom Leslie
· Patrick Lucas · Emily Makas · Igor Marjanovic · Andrew McLellan
· Mikesch Muecke · Gregory Palermo · Jorge Prado · Kiel Moe
· Marek Ranis · Rachel Rossner · Bryan Shields · Jen Shields
· Greg Snyder · Ann Sobiech- Munson
· Michael Swisher · Sean Vance · Nora Wendl
· Catherine Wetzel · Janet Williams · Peter Wong · Natalie Yates

SKETCHING WITH CODE: DEVELOPING PROCEDURIAL LITERACY IN EARLY ARCHITECTURAL EDUCATION
Making Virtual

NICHOLAS S. SENSKE, DOCTORAL CANDIDATE
UNIVERSITY OF MICHIGAN

Introduction

Programming does not have a good reputation in architecture. Older designers might remember having to learn FORTRAN, PASCAL, or some other programming language when they were in school. For most, it is not a fond memory. Early attempts at teaching programming to architects focused on tasks which were either too mundane (e.g. drawing and spreadsheets) or too esoteric (theory-driven applications such as shape grammars) to hold the students' interest. Besides, in a few years, programming seemed to be obsolete. When software with direct manipulation[1] interfaces became available it seemed to make more sense to push vertices around with a mouse than with code. Moreover, one didn't need to subscribe to a complex theory of design to do it. Most students who had to sit through these early courses never programmed again. [2]

But perhaps it is time to revisit the idea of programming in architecture. In the first half of this paper, I argue that basic computer programming has an important role to play in beginning design education. In the second, I propose a pedagogical framework for improving how it may be taught.

I. Procedural Literacy

While direct manipulation interfaces have made working with computers easier, they do not leverage the full potential of computation. Most architects today still perform much of their work by hand, drawing and updating every individual line and surface. But this may soon change. The next generation of design software involves *indirect* manipulation, specifying instructions, rules, and relationships so the computer can perform much of the mundane work itself. This is known as computational production and it has the potential to dramatically expand our capacity for mental and creative labor. It is already transforming other professions such as stock trading, biology, and journalism, among others, and is likely to do the same for architecture in the near future. Computational production is not an augmentation of existing practices, but a redefinition[3]; a different way of working than people are accustomed. Moving forward, it is likely to be the dominant method of architectural design. As such, it should be taught to students early in the curriculum, in parallel with other ways of making and considering design. However, in many architecture programs, one finds several examples of computational production – parametrics, generative design, dynamic architecture, data integration, etc. – taught as separate, advanced subjects. There is no provision made for a basic course in computation, a foundation in the concepts and mindset which should be prerequisite for these advanced labs and studios.

Mastering computational production involves learning a particular set of tropes and skills, but most importantly, adopting a different outlook. Computers are machines whose operation is defined by procedures. As such, the key to working well with computation is to understand process. For instance, most of the tools we work with are "black boxes". One can interact with the controls on the outside, but may not know how or why the tool works. With computation, these details are important. The operational logic of a computational system is often comprised of complex chains of cause and effect. Thus, one cannot make

1 Direct manipulation is the interface paradigm most users experience today. It involves interacting with graphic symbols (i.e. icons) through pointing and selecting.

2 McCullough, Malcolm. "20 Years of Scripted Space." *Architectural Design* 76 4 (2006): 12-15.

3 Pea, Roy D. "Beyond Amplification: Using the Computer to Reorganize Mental Functioning." Educational Psychologist 20 (1985): 167-82.

any assumptions about how a program works based on input and output alone. Understanding process, then, is critical to making sense of these systems. To cite another example, because computers can execute billions of procedures quickly and without error, they are capable of feats no human could achieve. Designers must learn to think and act at a different scale of production, beyond what they can touch or observe themselves. Last, designers are typically dependent upon others for their software tools. They are used to having the same tools as other designers and working under a set of inflexible limitations. But, with the proper procedural description, a computer can become nearly any tool. Taking full advantage of computation involves a faculty with abstraction, the ability to improvise with small programs as part of one's personal process.

Procedurality is a unique property of computers as a medium; what every computational artifact or technique has in common. To get the most out of their software and to recognize and overcome its limitations, designers need to be able to think procedurally: to write procedures to create effects and anticipate the effects of a given procedure.[4] Moreover, architects must be able to translate their knowledge of design into the realm of computation, considering how they design and even what design is. Without an understanding of process, designers are limited in their approaches and disadvantaged when learning computational tools. And so, students need to learn basic procedural literacy: how to read, write, and reason with procedures. To achieve this literacy, they must learn how to program.

While it may be possible to learn a kind of procedural literacy using analog means (studying cooking, for instance), transfer of knowledge from one domain to another is difficult.[5] Since students will be applying procedural thinking with computers, it makes sense that they learn it with computers. Moreover, instructions for a computer are different from those among humans. For example, computer code requires explicitness; human language is full of inference and assumptions.[6] In this sense, code is useful because it is a general language for describing process which is both human and machine-readable. However, the particular programming language studied is not important. Rather, the goal should be to learn the concepts and structures shared by all programming languages.[7] The expectation is not for students to become software developers. Procedural literacy is just that; literacy. While most people know how to read and write, not everyone is a professional novelist. But like writing, designers should learn programming in order to be able to express themselves, to navigate their culture, and, most importantly, to think.

An early course in programming, which helps train students to work with process, may serve as a useful foundation, something that will have relevance despite changes in technology. The challenge is that learning programming is difficult. By a rough estimate, nearly 35% of computer science students drop out – even in the best programs.[8] Of those who do graduate, many lack a basic understanding of programming concepts.[9] Some would believe that programming is hard because it depends upon humans writing (seemingly) cryptic code. They argue that a better language or graphical interface is the solution. But the details of programming languages don't present a problem for novices very long. Even young children can master them, given enough time.[10] After a semester, syntax is no longer a problem for

4 Sheil, B.A. "Coping with Complexity." *Information Technology & People* 1 4 (1983): 295 - 320.

5 Perkins, D. N., and Gavriel Salomon. "Are Cognitive Skills Context-Bound?", 1989. 16-25. Vol. 18.

6 Larsen, SF. "Procedural Thinking, Programming, and Computer Use." Proceedings of the NATO Advanced Study Institute on Intelligent Decision Support in Process Environments. Ed.

7 Mateas, Michael. "Procedural Literacy: Educating the New Media Practitioner." *On The Horizon.* Special Issue. Future of Games, Simulations and Interactive Media in Learning Contexts 13 1 (2005).

8 Guzdial, Mark, and Elliot Soloway. "Computer Science Is More Important Than Calculus: The Challenge of Living up to Our Potential." ACM, 2003. 5-8. Vol. 35.

9 Clear, Tony, et al. "The Teaching of Novice Computer Programmers: Bringing the Scholarly-Research Approach to Australia." Tenth Australasian Computing Education Conference (ACE2008). Ed.

10 Kay, Alan. "The Early History of Smalltalk." ACM SIGPLAN Notices 28 3 (1993): 69-95.

most users. It is the procedural errors and the design of procedures that remain an issue.[11]

While better tools can help eliminate unnecessary details and connect computational ideas to domain knowledge, they can't eliminate the thinking required. As Michael Mateas points out, even with the perfect interface— if we could simply tell the computer what we wanted to do – we would still need to be able to design and describe procedures. No matter how intelligent the software, "expressing ideas will always take work".[12] Procedural thinking won't emerge spontaneously from better tools. The problem with learning programming is not technological, it is psychological and cultural.[13] The solution must be educational.

The fact is that students don't learn enough about process in traditional programming and digital media courses. Instead, these courses tend to focus on the surface details of code, the syntax and commands.[14] These details are necessary but not sufficient for procedural literacy. In addition, students are often taught computational tropes using rote tutorials. While following tutorials enables them to attempt more sophisticated projects, the knowledge they learn is brittle. If a student encounters a context which is different from the original tutorial, they may not be able to recall the technique or apply it properly. Moreover, being given the steps to implement something is not the same as deriving those steps oneself. Tutorials do not teach students how to design their own procedures or why the procedures within the tutorial are structured a certain way. Students need commands and patterns, but they also need a higher order framework for making sense of them in the context of their work. This is what is missing from most pedagogy of computational production.

How can architects learn procedural literacy? Perhaps how they already learn visual literacy. Sketching is one of the first courses in the architectural curriculum; a foundation for all courses to follow. It teaches architects how to draw, but more importantly, how to think about form and design. Its rigorous nature and progression from concrete to abstract concepts promotes the development of a robust mental model for representation. As such, I propose that we ought to teach programming like a sketching class.[15]

II. Sketching in Code

Instead of instructing students how to operate a programming language as one might an industrial tool, educators should teach computation as a flexible medium for thinking. In the remainder of this paper, I will detail how "sketching with code" might serve as a model for achieving such a goal.

Motivation

Most people find programming intimidating. In my experience, designers often have anxiety about learning it because they don't consider themselves proficient in math and logic.[16] At the very least, they believe programming falls outside of their profession. It is important to address this anxiety early because how a person feels about what they learn can be as important as how they are taught. To a certain extent, students will do whatever is asked of them, but if they lack confidence in themselves and are uninterested in the material, their experience is less likely to be productive.

We learn best when we are in an environment in which we feel capable and supported. Consider how gymnasts practice their routines with guide ropes, pads, and nets. Because they are less afraid of getting injured if they fall, they can place more of their effort on improving their performance. Similarly, in design education, sketching class is a safe environment

11 Linn, Marcia C. "The Cognitive Consequences of Programming Instruction in Classrooms." 1985. 14-29. Vol. 14.

12 Mateas, Michael. "Procedural Literacy: Educating the New Media Practitioner." *On The Horizon*. Special Issue. Future of Games, Simulations and Interactive Media in Learning Contexts 13 1 (2005).

13 Sheil, B.A. "Coping with Complexity." *Information Technology & People* 1 4 (1983): 295 - 320.

14 Soloway, E. "Learning to Program = Learning to Construct Mechanisms and Explanations." ACM, 1986. 850-58. Vol. 29.

15 The sketching metaphor is not my own invention. It is part of a tradition of pedagogical programming languages such as Processing, Design by Numbers, and Logo, which are designed to enable users to create visual forms with a minimal amount of code. I take the efforts of these languages and their creators as a pedagogical jumping-off point.

16 Nor would they want to be, as those things seem like the very antithesis of creativity to most designers.

in which one can learn to draw. There is no expectation of perfection. The sketchbook is a place to try things, to repeat them, and to fail without penalty. False-starts and mistakes far outnumber one's "good" sketches. And that's okay. In sketching class, students might lack self-confidence at first, but they are willing to try. This is the earnestness we ought to duplicate in an early programming course.

Calling programs sketches, although it is a small gesture, can help ease students' apprehension. As a metaphor, it connects what they are doing to architecture and sets the expectation that their programs will be short and rough (see: Practice). If students know they aren't expected to be great programmers right away, they may be more willing to suspend their fear and make an effort.

Practice

To learn a craft — to develop skills and an intuition for a medium — demands a fair amount of hands-on practice. A sketching class revolves around this notion. Students draw and they redraw. Repetition and refinement is the order of the day. They fill entire sketchbooks with the shared understanding that their goal is not a well-refined piece, but rather learning how to draw.

A first programming course should be a similar experience. But instead, students are introduced to programming in advanced labs or studios where they might only implement a few programs over the course of a semester. This is simply not enough practice, and of insufficient variety, to get a feel for the complexities and contradictions of procedural work. The typical pedagogy of lengthy tutorials and multi-week projects implicitly emphasizes product over process; following instructions and getting something to work (by any means necessary) rather than understanding how it works. Students may write programs, but they do not necessarily learn how to program.

As with any craft, the best way to learn to program - and to learn from programming - is to do a lot of it. Like a sketching class, an introductory programming course ought to focus on a rigorous sequence of small exercises designed around the fundamentals of the medium. I taught a course at the University of Michigan last fall[17] with this idea which I cite as one example of how to implement this in the classroom.

In a typical hour of my course, I had students write as many as eight to ten small programs in Processing. That might sound like a large number, but these "sketches" consist of only a few lines of code. With careful planning, a sketch can produce sophisticated and interesting visual output which illustrates the concept at hand. Because the programs are so short, students have an easier time following the flow of the code. Also, if a student makes a mistake or has a misunderstanding, it can be diagnosed quickly. Like a drawn sketch, these programs are not expected to be efficient or flawless, but rather an opportunity to learn.

A traditional programming lecture might demonstrate the same number of examples as my class in the same amount of time, but I believe there is a benefit to having students type the code and observe the results for themselves. The experience of coding engages more senses and is more involving than merely watching the instructor. Once students have made their sketch, they can experiment and try different options on their own, testing the limits and potential of the concept. They can't do this if the instructor is merely showing the example to them (and many of them won't do it at all outside of class). This also gives them a bit of room for creativity, which can be more motivating than following along with fully-prescribed examples.

Ultimately, my students wrote far more programs than they might in a typical programming class. While a student working on a tutorial or a studio project might be stuck debugging the same handful of loops, a student in a sketching class, as in my example, could write and experiment with dozens of loops across a multitude of contexts. In my experience, increasing students' practice time gives them a more robust understanding of computational concepts – where and when to apply them; exceptions, etc. – and helps them grasp the medium as a whole.

17 Course website at: http://arch506-f09.tcaup.umich.edu/

Cognitive Loading

Many computation courses involve too much design. Students are expected to learn programming and apply it fluently at the same time. Even in an advanced course, this is unreasonable.

Abstraction and synthesis cannot occur while one is still learning to comprehend the medium. It's like learning to drive a car. At first, there are so many unfamiliar details to monitor — steering, gas, signals, etc. — that navigating the vehicle to a destination is often more than a person can handle. Until the new driver is comfortable with the controls, they aren't going anywhere.

In cognitive science, this idea is known as cognitive loading. The more things one has to keep in active memory, the more difficult it is to perform well. One of the reasons programming is so challenging is because it has a substantial cognitive load.[18] Even in basic programs, there are many elements to keep track of: proper syntax, remembering commands, program flow, variable states, and so forth. Adding design (which is also a complex task) to the mix may be asking too much of novices.

With traditional sketching, the constraints of the course allow students to gain familiarity with the nuances of the medium. Students are not expected to think up original work or innovative methods. As such, they can focus on developing skills and learning a set of principles from drawing which they can apply to form and design. In a basic programming course, the same idea should apply.

Cognitive loading extends to lesson plans, as well. Too many details or prerequisites and students can easily become lost and confused. To alleviate this, it can be helpful to remove any unimportant details that might slow students down, especially when introducing a new concept. For example, having students write their programs from scratch might be realistic, but for novices it introduces more details to track and opportunities for errors. Students might so much spend time and effort typing and correcting punctuation errors, that they become distracted from the main idea of the lesson. To help reduce cognitive load, instructors can prepare programs ahead of time and have students modify or add small sections to them. This method of turning complex programs into simplified sketches makes it possible to cover more material with greater depth.

Although programming will never be as simple as sketching with a pencil and paper, to teach it well, we ought to be wary of its complexities and seek to reduce them wherever we can.

Transfer

An important goal of learning is to be able to apply knowledge and skills learned in one context to other situations. In education this idea is known as transfer[19]. With procedural literacy, the hope is that computational concepts and thinking skills will transfer to any software or design challenges students may face.

The problem is that most programming courses do not successfully promote transfer. As discussed earlier, many of them emphasize surface details of the code and depend upon rote tutorials. These activities often make knowledge inert – locked within the context in which it was learned.

In education, there appears to be an implicit assumption that transfer happens on its own. For example, if students are immersed in writing code long enough, eventually they will figure out how to think procedurally. Research has shown consistently that this is rarely the case. To encourage transfer, it is best to teach with transfer in mind.

Sketching courses tend to do this well (although they may not invoke the theory of transfer when doing so). In these basic studios, it is understood by both teachers and students that more is being taught than merely how to draw. This is important because one of the keys to transfer is priming the student – preparing them to see beyond the surface details and mindfully abstract what they learn. Students know that sketching is not the end, but the means. Drawing is almost secondary to learn-

18 Guzdial, Mark. "How We Teach Introductory Computer Science Is Wrong." Communications of the ACM (October 8, 2009). Accessed January 10, 2010 <http://cacm.acm.org/blogs/blog-cacm/45725-how-we-teach-introductory-computer-science-is-wrong/fulltext>.

19 For a good survey of this topic, see Butterfield, Earl C., and Gregory D. Nelson. "Theory and Practice of Teaching for Transfer." Educational Theory Research and Development 37 3 (1990): 5-38.

ing the basic concepts of representation and form.

How does one actively teach for transfer? There are two commonly discussed methods. First, there is low road transfer, in which the learner practices an activity extensively and deliberately in a variety of situations to the point of near automaticity. Essentially, one over-learns something to the point where they develop a behavioral response, an intuition. But, to be clear, few courses are designed to involve students in the amount of effort this takes. In order for this kind of transfer to occur, it can take a considerable amount of time.[20]

Second is high road transfer, in which lessons are designed to promote a deliberate abstraction of principles. For instance, students might be given several related examples and then asked to come up with a principle they share. Later, the same students would be asked to determine if the principle applies in a series of different situations. In this manner, the students' knowledge is effectively decontextualized, made and not merely given.[21] The trouble with this method is that the material must be presented in a highly specific way in order to trigger transfer. Once again, most programming courses do not have this kind of structure.

I believe sketching accomplishes transfer as a combination of both methods. Students draw a great deal, practicing to develop hand-eye coordination but also internal generalizations about types of form and visual and aesthetic principles. In addition, students' sketches are used by the class to explicate and examine principles. It is this structure, which promotes both low and high road transfer, that I believe programming classes ought to emulate.

Feedback

Imagine you are a golfer trying to improve your game. Hoping to fix your swing, you drive a few balls while your golf pro watches. A week later – while you are putting – she calls back and explains why your shots tend to hook. How much do you think the pro's advice will improve your drive? Probably not much. And yet, this is analogous to the kind of feedback many designers receive in programming courses.

Most students' only practice with coding is outside of class, in their homework or projects. Because the grading process can take so long, there can be a considerable lag between when they submit their work and receive comments. Often, students won't hear back about their programs until after the next lesson. By this point, they have likely shifted their attention to the new material. They have little motivation or incentive to return to the old work and correct their mistakes.

In a traditional sketching class, students receive active coaching while practicing. Feedback is frequent and timely. The teacher walks around as students draw, assisting and making comments. The low response time between practice and feedback is beneficial to helping students correct their performance. Basic behavioral psychology tells us that reinforcement occurs when treatment closely follows an action. The sooner a student receives coaching, the more likely they are to correctly interpret the material. Ideally, coaching would occur while they are engaged in a task.

The quality of feedback is also important. For example, in a typical programming lab, students may receive timely help, but it is seldom constructive. When the lesson consists of following tutorials, feedback from the instructor is not focused on an individual's understanding, but on making sure everyone completes an instruction so the class can move on to the next one.

In contrast, a sketching exercise, which does not require a sequence of interdependent steps, allows for a greater flexibility of pacing. As such, the instructor can steer students towards comprehension rather than compliance – correcting any misunderstandings in their mental model of the medium.

Sustained practice is essential to developing skills and understanding, but repetition alone is not enough. Feedback, at the right time and of the proper type, is essential to making practice worthwhile.

20 Perkins, D. N., and Gavriel Salomon. "Are Cognitive Skills Context-Bound?", 1989. 16-25. Vol. 18.

21 Perkins, D. N., and Gavriel Salomon. "Teaching for Transfer." Educational Leadership 46 1 (1988): 22-32.

Conclusion

Computational production is reshaping professions. To adapt and thrive, designers will need procedural literacy. They must learn – and learn from – programming. Unfortunately, teaching programming successfully is a challenge. While most students can pick up a language, they often fail to learn procedural thinking. The "sketching with code" framework, described in the second half of this paper, is an attempt to address the shortcomings of traditional programming and digital media courses and steer students towards procedural literacy.

The components of the framework: improving students' motivation, reducing task load, teaching for transfer, and providing timely feedback, are not new ideas in education. One could say this is simply what good teachers do. However, I have found through my research and in my own experience that these elements don't often come together in programming classes and this –not poor student aptitude or unintuitive tools— is the reason why most students are unsuccessful.

It is my hope that sketching might serve as a familiar metaphor for design educators; a reminder as to what the purpose and method of teaching programming ought to be. We sketch in order to think and such thinking cannot be reduced to (or induced from) rote instructions. It must be coached and cultivated through deliberate practice over time. The reverence and patience we reserve for teaching students drawing ought to be applied to our curriculum for computation. As manual sketching is to CAD plans and 3D models, so is basic programming to the future of the profession.

Works Cited

Butterfield, Earl C., and Gregory D. Nelson. "Theory and Practice of Teaching for Transfer." *Educational Theory Research and Development* 37 3 (1990): 5-38.

Clear, Tony, et al. "The Teaching of Novice Computer Programmers: Bringing the Scholarly-Research Approach to Australia." Tenth Australasian Computing Education Conference (ACE2008). Ed.

Guzdial, Mark, and Elliot Soloway. "Computer Science Is More Important Than Calculus: The Challenge of Living up to Our Potential." ACM, 2003. 5-8. Vol. 35.

Guzdial, Mark. "How We Teach Introductory Computer Science Is Wrong." Communications of the ACM (October 8, 2009). Accessed January 10, 2010 <http://cacm.acm.org/blogs/blog-cacm/45725-how-we-teach-introductory-computer-science-is-wrong/fulltext>

Kay, Alan. "The Early History of Smalltalk." ACM SIGPLAN Notices 28 3 (1993): 69-95.

Larsen, SF. "Procedural Thinking, Programming, and Computer Use." Proceedings of the NATO Advanced Study Institute on Intelligent Decision Support in Process Environments. Ed.

Linn, Marcia C. "The Cognitive Consequences of Programming Instruction in Classrooms." 1985. 14-29. Vol. 14.

Mateas, Michael. "Procedural Literacy: Educating the New Media Practitioner." *On The Horizon*. Special Issue. Future of Games, Simulations and Interactive Media in Learning Contexts 13 1 (2005).

Pea, Roy D. "Beyond Amplification: Using the Computer to Reorganize Mental Functioning." *Educational Psychologist* 20 (1985): 167-82.

Perkins, D. N., and Gavriel Salomon. "Teaching for Transfer." *Educational Leadership* 46 1 (1988): 22-32.

Perkins, D. N., and Gavriel Salomon. "Are Cognitive Skills Context-Bound?", 1989. 16-25. Vol. 18.

Sheil, B.A. "Coping with Complexity." *Information Technology & People* 1 4 (1983): 295 - 320.

Soloway, E. "Learning to Program = Learning to Construct Mechanisms and Explanations." ACM, 1986. 850-58. Vol. 29.