THIS PAGE LEFT
INTENTIONALLY
BLANK

A Curriculum for Integrating Computational Thinking

Nick Senske

University of North Carolina at Charlotte

For architectural educators, a challenge of teaching digital design is maintaining a relevant curriculum amidst an increasing array of constantly evolving software and tools. This paper describes a curriculum proposal under review at the University of North Carolina at Charlotte, which attempts to address this situation through the integration of computational thinking in studios and seminars.

Computational thinking is a developing area of study that originates from the discipline of computer science. Researchers define it as the ability to leverage the strengths of computing in order to study and solve problems. Whereas most digital design education is narrowly focused on the acquisition and application of a limited set of software commands and techniques, computational thinking is concerned with the fundamental concepts at the heart of every piece of computational software and hardware. In future testing, we hope to demonstrate that a more holistic mindset, rooted in these first principles, can assist students with understanding and making effective use of any form of computing they may encounter— now or in the future.

Introduction

At the University of North Carolina at Charlotte (UNCC), the School of Architecture is developing a new integrated curriculum for digital design. The primary impetus for this effort is the growing professional demand for students proficient in the latest software methodologies, namely parametric design and Building Information Modeling. Learning these tools involves more than merely learning a new interface; it requires a change in one's thinking about design. In addition to representing buildings visually, with digital drawings and models, designers need to understand how to represent them symbolically, as dynamic systems of rules and relationships. While this is a significant shift in how architects conceive of their work, it is only the beginning of a growing trend towards computational design.

As our faculty considered the reorganization it would take to keep our courses up-to-date, we wondered: should our program be looking ahead to the future, towards scripting or other forms of programming? What about data visualization and simulation? Given the pace of advancement in our discipline, how long would our new curriculum remain relevant?

Adapting to changes in technology can be challenging for educators. The tools and methods of digital design evolve quickly, while the pace of administration tends to lag behind. What meaningful lessons can we teach our students about digital design that will not be rendered obsolete by next year's software or the new tools just over the horizon?

In this paper I describe a proposal under review at UNCC that seeks to address this issue of relevance. I will argue that teaching computational thinking can provide students with a lasting foundation; a mindset that is compatible with design, while taking advantage of the most powerful aspects of computing. We believe this new curriculum will overcome flaws in current digital pedagogy, resulting in students who are more engaged and adaptable with digital media.

Problem

In the book Design by Numbers, John Maeda argued that "we implicitly glorify rote memorization as the basis of skill for a digital designer" [1999]. Many students' understanding of software is limited to sequences of commands. As a result, they have a superficial grasp of digital design. Instead of deriving solutions from principles and structure, they look at the surface of problems to determine which sequence to use. When problems arise in the software, they have no mental model of the system to organize their thinking, and so they engage in haphazard "hacking" behavior to seek out a solution. They suffer from what Roy Pea describes as "production without comprehension"[1983].

These difficulties can be amplified when students attempt to learn computational design. There is far more to account for in the creation of a dynamic logic-driven system than in producing a static representation. Learning the sequences of commands to make the system – as one might in a lecture, tutorial, or textbook – does not teach one how or why the steps go together in that particular order. In other words, the command knowledge teaches nothing about design or problem solving.

Furthermore, completing additional computational tutorials does not necessarily result in a more comprehensive outlook. A correct mental model of a computational system is difficult to infer based on surface details. This is because computational systems do not work in simple cause-and-effect relationships. Effects from one part of the system can propagate to others without any visible effect [Sheil, 1983]. For this reason, the models that users infer from computational systems are often full of misconceptions and outright errors.

A student's incomplete understanding can lead to designs that are inelegant, inefficient, or do not work. Most often, however, it produces designs that are severely limited; example or tutorial solutions with a few minor variations. Learning more tutorials increases the effectiveness of this tactic, but does not solve the problem of comprehension.

Moreover, when the problem or the software changes significantly enough, rote patterns no longer apply.

There is a significant gap in digital design classes that occurs between teaching commands and critiquing designs. In order to improve students' comprehension of the material, educators need to do more than help them collect knowledge about tools. Studies of performance show that experts don't just know more commands or patterns; they think differently about problems. They have robust mental models of their working domain and a perspective based on principles, not surface details [Dalbey and Linn, 1985]. The question is: how can architectural educators produce this kind of deeper learning with regards to computation? What goes into the gap?

Computational Thinking

In our new curriculum, we propose to address these challenges by teaching computational thinking. Computational thinking is not a new idea or one that is unique to architecture. It is a growing area of research that originates from the discipline of computer science [Wing, 2006]. Broadly defined, it is the ability to leverage the strengths of computers in order to study and solve problems. Computational thinking involves knowledge of the fundamentals of computing such as abstraction, iteration, and data structures. In addition, it includes the skills to break down a problem or task in terms of the computational steps involved, to test solutions-in-principle and refine them based on feedback, and to use them to study and solve problems. It also refers to a sense of procedural reasoning: the ability to envision the structure of procedural systems and anticipate their outcomes [Sheil, 1983]. When architects talk about the necessity and difficulty of "thinking differently" with tools like BIM, this is likely the gap. Explicit instruction in computational thinking is precisely what is missing from most classrooms and textbooks.

There are significant advantages to computational thinking besides learning to use the latest tools well. First, all forms of computing – all software, all programming languages, and devices – can be understood as different instances or interfaces of computation [Blackwell, 2002]. Therefore, a person that understands computational principles has a basis for making sense of anything computer-based. The tools might change, but the underlying logic does not . Second, software and devices are used most effectively when they take advantage of computation [Crawford, 1987]. Third, understanding how

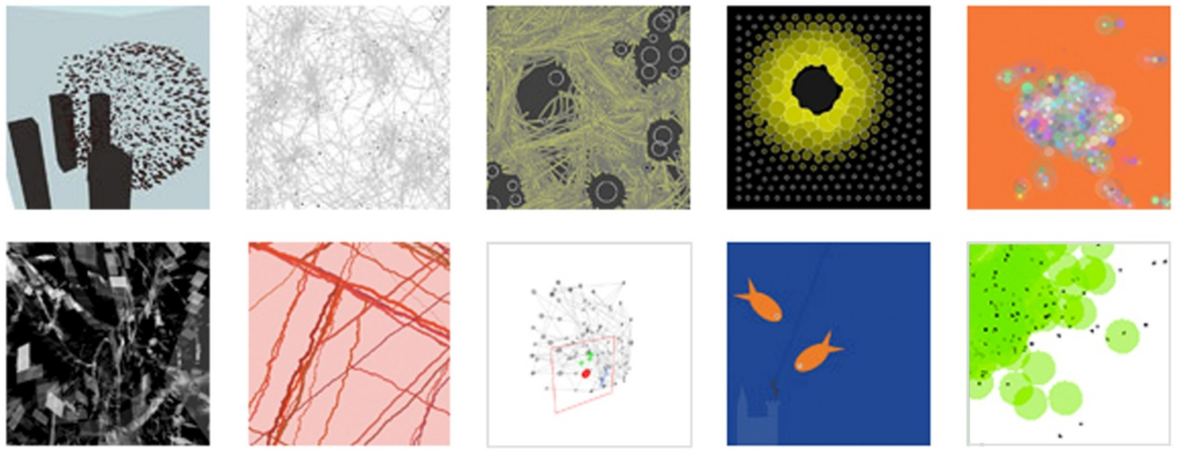


Figure 1. Student examples of computational design. From a pilot study of Computational Methods, conducted by the author at the University of Michigan, Ann Arbor in 2009.

to exploit these capabilities allows one to represent and solve problems computationally. This has the potential to redefine one's work [Pea, 1985]. For instance, the way that algorithmic stock trading and "shotgun" gene sequencing transformed the fields of business and biology, respectively. In the words LOGO (and LEGO Mindstorms) inventor, Seymour Papert, computational thinking allows us to "get someplace different" [1993].

To illustrate these ideas in practice, consider an example that everyone has some experience with: Microsoft Word. Most people use Word for little more than typing, formatting, and printing documents. In this scenario, the processing power of the computer is not fully exploited. The software does not substantially change the way one approaches the task compared to a typewriter. However, applying computational thinking, one might recognize that the document is more than a simulation of paper, but is actually a data structure. Instead of editing a repeated mistake by hand, users would apply the principle of substitution and seek out the Find and Replace command. Similarly, one could approach formatting parametrically, designing and associating Styles with elements in the document. Writing a form letter, one would invoke the principle of propagation and use the Mail Merge command to associate data from a spreadsheet for words in the document. These functions save time and help reduce mistakes, but most novices, who operate the program based on surface characteristics, do not know they exist and would not think to look for them. And so, with computational thinking, one does not have to approach the tool as its metaphors might suggest, but from an understanding of what computers do well. A person who uses it this way is almost certainly using it more effectively.

But more than this, a computational perspective can completely change the potential of the software. Returning to the previous example, Malcolm McCullough once described to me how he uses Word as a research tool, counting the number of times an author used important words or phrases. This simple frequency analysis helps him find patterns in arguments and citations. Now, take things a step further and visualize this analysis. A simple script that associates frequency to font size produces a tag or word cloud[1] – a quick way to summarize a body of text. So thinking computationally, even about something as mundane as Word, can transform our way of thinking about the tool and lead to the creation of new knowledge.

Computational thinking is empowering. A student that understands the principles of computation in a program like Word should be able to recognize them in Photoshop, AutoCAD, Grasshopper, and even Python. While the specific uses and interfaces of these programs and languages are different, the ideas of computation, like substitution, parameters, and propagation can be found in all of them. Knowing principles makes it easier to learn the tool and use it more effectively. In addition, a student who can think in terms of these principles will be better prepared for any new program they might encounter post-graduation. And so, learning computational thinking has more lasting educational value compared to other methods of teaching digital design software.

Curriculum

Unfortunately, no one knows exactly how to teach computational thinking. This is a well-known problem even in computer science. Studies from this field suggest that it is highly unlikely that architecture students could learn it simply by taking a programming course or one that specifically teaches thinking skills. Without special effort on the part of instructors and learners, novices who study a programming language tend to demonstrate poor performance, particularly in the design of programs [Soloway 1986]. Nor do they gain additional benefits such as improved problem solving skills -- something we might expect as part of computational thinking [Mayer 1986]. These findings imply that developing computational thinking in designers requires a different approach.

The challenge facing educators who want to teach

[1] See http://www.wordle.net/

thinking skills is one of transferable knowledge[2]. This is knowledge applied outside the context in which it was learned-- the very opposite of rote learning. There are two basic types of transfer, known as near and far transfer. We are interested in both. Consider a student who learns a particular technique for procedurally generating a roof panel system. If the student understood that the paneling method could be used to make floor and wall systems in the same software, this would demonstrate near transfer of learning. If the same student applied the logic of the paneling system in a GIS scripting language to distribute plots for a real estate development, this would be evidence of far transfer. And so, transfer is desirable because it makes learning more efficient for both teachers and students. If the goal is to graduate students who can apply programming ideas to design and adapt to changes in technology, teaching for transfer should be a priority.

Transfer rarely happens spontaneously, even with significant practice [Perkins and Salomon 1989]. Procedural knowledge, such as following instructions to use a program, is especially prone to rote learning. It takes a specific kind of teaching to break free of this. As such, in order to help students become flexible computational thinkers, our proposed curriculum uses teaching methods and course materials that have been proven to help with transfer. We design our lessons with three basic requirements in mind. First, software demonstrations need to take into account many contexts. This is so that students get a sense of when a particular technique or idea applies, and whether there are any exceptions to this. Most tutorials rarely explore enough variations for this to occur. Second, and most critically, transfer must happen mindfully [Salomon 1985]. This is to say that students must be guided and encouraged to extract the principle or strategy themselves. They must study examples intently instead of stepping through them in a disassociated manner. In this way, the student takes ownership of the principle. They are not simply told it; they discover it. Lastly, transfer depends upon metacognition or "thinking about thinking" [Perkins and Salomon 1988]. The student needs to be taught how to self-monitor, to examine the kind of task they want to perform, determine which principle applies, and adapt or correct their approach if it doesn't seem to be working. Each of these requirements takes extensive preparation and careful classroom management, but the net effect is to transform learning digital techniques and methods from a passive experience to an active one.

Another way our proposal supports transfer is in the sequencing of courses within the curriculum. We do not propose one class that teaches computational thinking. Rather, this knowledge is spread across several years of schooling. Research and experience suggest that it takes time and practice to develop this kind of thinking. Moreover, we want to show that computation is not limited to a single semester or to certain pieces of software. Rather, it is a way of doing things that is compatible with design thinking. If students see computation in multiple contexts and in ways that connect to their level of understanding, they will be more likely to integrate it into their work.

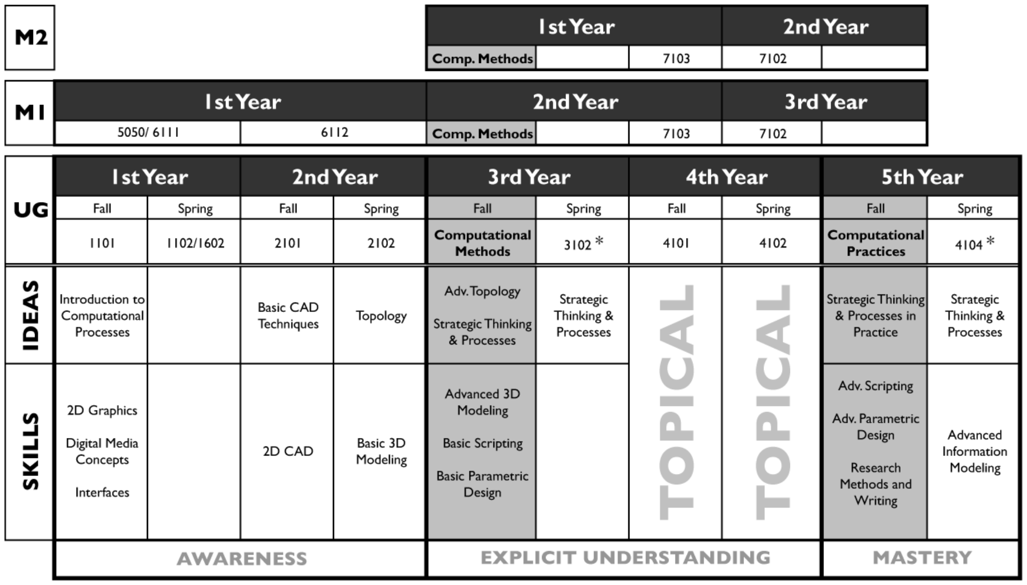
Our course sequence is based on an educational methodology called cognitive apprenticeship [Collins et al, 1990]. This is similar to the idea of design studio, in which students encounter a successive approximation of how professional designers think and work. In our case, we hope to inculcate a model a mature computational thinker.

The curriculum divides the teaching of computational thinking into three different stages of apprenticeship. In their second year, students encounter what we call the “awareness” phase. The objective of this phase is to introduce students to computation early in their education, in a way that is motivating and interesting, but not so deep that they find the material overwhelming. In our design studios, students are taught computational strategies while they learn design software like Photoshop, AutoCAD, Rhinoceros, etc. The content of these lessons is similar to the earlier Word example. We

[2]For a review of literature on transfer, see [Butterfield, et al 1990]

show students how approaching software computationally makes them more effective designers. This serves two purposes: 1.) it gives computational ideas immediate relevance and 2.) it destabilizes the idea that computation is limited to programming or software such as Revit and Grasshopper.

commands and so are not easy for a person to discover on their own. Moreover, they are general enough to apply to nearly any kind of software. Bhavnani's research demonstrates that students who learn strategies are likely to apply them in new contexts.



M2					1st Year			2nd Year		
					Comp. Methods		7103	7102		
M1	1st Year				2nd Year			3rd Year		
	5050/ 6111		6112		Comp. Methods		7103	7102		
UG	1st Year		2nd Year		3rd Year		4th Year		5th Year	
	Fall	Spring	Fall	Spring	Fall	Spring	Fall	Spring	Fall	Spring
	1101	1102/1602	2101	2102	Computational Methods	3102 *	4101	4102	Computational Practices	4104 *
IDEAS	Introduction to Computational Processes		Basic CAD Techniques	Topology	Adv. Topology Strategic Thinking & Processes	Strategic Thinking & Processes	TOPICAL	TOPICAL	Strategic Thinking & Processes in Practice	Strategic Thinking & Processes
SKILLS	2D Graphics Digital Media Concepts Interfaces		2D CAD	Basic 3D Modeling	Advanced 3D Modeling Basic Scripting Basic Parametric Design				Adv. Scripting Adv. Parametric Design Research Methods and Writing	Advanced Information Modeling
	AWARENESS				EXPLICIT UNDERSTANDING				MASTERY	

Figure 1. Provisional curriculum diagram, University of North Carolina at Charlotte, 2010.

The strategies are based upon a framework developed by Bhavnani, et al [2008]. Compiled through a careful study of expert users, their framework teaches efficient and effective uses of software that are derived from the powers of computation. An example of a strategy is "reuse and modify groups". This idea comes from the principles of abstraction and iteration. For instance, when drawing a series of doors, it is faster to make one door and to copy and modify it than it is to draw each door separately. Similarly, making a spreadsheet, it is faster to drag a formula across cells and then modify it than to write multiple specific formulae. This is a basic example, but something than many novices and even experienced users tend to overlook. Strategies like this are interesting to students because they save time compared to other methods. They are not learned by studying software

In the third year of the curriculum (the middle year for most of our students) we position our keystone class, Computational Methods. This is what we call the "explicit understanding" phase. It marks the transition from peripheral usage of computation to intensive practice. This seminar-style class surveys the principles that underlie computation: variables, conditionals, data structures, etc. Students are not merely shown or taught the principles. Instead, students derive them through rigorous exercises that relate to design. To cite two examples, one lesson teaches students about logic and algorithm design by having them write generative rules for Bauhaus compositions. In another lesson, they hand-code simple parametric systems from scratch. The later example could be done with software like Revit, but by making the steps involved explicit, the programming exercise helps the student grasp what is happening computationally as well as the design thinking that goes into creating a parametric object in software.

In the first half of the course, students study the principles using a scripting language such as Processing or Python. Learning to write code is important for architects because most design tools have built-in scripting languages that allow users to automate tasks or associate their information with other programs. Moreover, programming skill addresses a broader domain than drawing and modeling. It connects to other media such as visualization, simulation, physical computing, as well as fields outside of architecture.

The second half of the class reexamines the principles through computational software: parametric programs like Grasshopper and 3D-specific scripting languages like Rhinoscript. In most architecture programs, this would be students' first exposure to computation. Computational Methods treats these tools as another way to interface with computation and an opportunity to learn the advantages and disadvantages of representing processes in different ways.

Lastly, in their final year, students move into the mastery phase. The goal of this phase is for students to transition to independence, applying what they have learned about computation towards self-directed inquiry and reflection. In a course we call Computational Practice, students work on projects with real objectives, such as competitions and conference papers, in groups and with people from other disciplines. We see this as a laboratory where students can cultivate a personal take on computation, developing a sense of their own process before they graduate into professional practice.

The School of Architecture at UNC Charlotte sees computation as an integral part of 21st century architecture. However, a reliable model for teaching this outlook does not yet exist. To address this challenge, the curriculum committee believes that computer science and educational research can provide valuable insights into successfully teaching computational thinking where intuitive, command-based pedagogies fall short. We forward to testing our hypothesis when the proposed curriculum debuts in the coming fall.

Evaluation

Because the curriculum is still under review, we have no assessment data to share at this time. We are presently collecting control data, with the expectation of following up when the new curriculum launches in the fall semester of 2011. Our assessment plans include both quantitative and qualitative components. In our quantitative tests, we will be looking for evidence of near and far transfer. Respectively: whether students are able to apply computational strategies and principles to effectively use programs they already know; and whether this knowledge helps them to use a program they have never seen before. These tests will be conducted using a protocol involving scripted problems, screen recording, and self-reporting by the subjects. The qualitative half of the assessment will use pre- and post-course surveys, and attempt to gauge students' values, attitudes, and understanding of digital media concepts. Our plan is to conduct these tests in a long-term study, repeating them which each incoming class, and following students' progression through the curriculum over several years.

Conclusion

This paper describes an integrated curriculum aimed at teaching students to be more mindful and flexible users of design computing. Rather than software training using rote tutorials, we propose a framework through which students might learn computational thinking: the fundamental principles of computation as they apply to problem solving and design. This framework, based on research from cognitive science and computer science education, consists of a sequence of courses intended to help students experience, identify, and demonstrate key computational principles. Our department's goal is for students to be able to apply computational thinking towards any current or future software they might encounter. As of this writing, the curriculum is still under review, but the first stages of our evaluation effort are already underway. We plan to report back on our findings in a future paper.

References

Bhavnani, S. K., Frederick A. Peck, and Frederick Reif. "Strategy-Based Instruction: Lessons Learned in Teaching the Effective and Efficient Use of Computer Applications." *ACM Transactions on Computer-Human Interaction* 15, no. 1 (2008).

Blackwell, Alan F. "What Is Programming?" In *14th Workshop of the Psychology of Programming Interest Group*. Brunel University (2002).

Butterfield, Earl C., and Gregory D. Nelson. "Theory and Practice of Teaching for Transfer." *Educational Theory Research and Development* 37, no. 3 (1990): 5-38.

Collins, A., John Seely Brown, and S.E. Newman. "Cognitive Apprenticeship: Teaching the Crafts of Reading, Writing, and Mathematics." In Knowing, Learning, and Instruction: Essays in Honor of Robert Glaser, edited by L.B. Resnick. Lawrence Erlbaum, (1990): 453-94.

Crawford, Chris. "Process Intensity." *Journal of Computer Game Design* 1, no. 5 (1987).

Dalbey, John, and Marcia C. Linn. "The Demands and Requirements of Computer Programming: A Literature Review." *Journal of Educational Computing Research* 1, no. 3 (1985): 253-74.

Linn, Marcia C. "The Cognitive Consequences of Programming Instruction in Classrooms." *Educational Researcher* 14, no. 5 (1985): 14-29.

Maeda, John. Design by Numbers. Cambridge: MIT Press (1999).

Mayer, Richard E., Jennifer L. Dyck, and William Vilberg. "Learning to Program and Learning to Think: What's the Connection?" *Communications of the ACM* 29, no. 7 (1986): 605-10.

Papert, Seymour. *Mindstorms: Children, Computers, and Powerful Ideas*. Cambridge: Perseus Publishing (1993).

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

Pea, Roy D. "Logo Programming and Problem Solving. [Technical Report No. 12.]." In American Educational Research Association Symposium. Montreal, Canada (1983).

Perkins, D. N., and Gavriel Salomon. "Are Cognitive Skills Context-Bound?" *Educational Researcher* 18, no. 1 (1989): 16-25.

Perkins, D. N., and Gavriel Salomon. "Teaching for Transfer." *Educational Leadership* 46, no. 1 (1988): 22-32.

Salomon, Gavriel. "Information Technologies: What You See Is Not (Always) What You Get." *Educational Pyschologist* 20, no. 4 (1985): 207-16.

Sheil, B.A. "Coping with Complexity." *Information Technology & People* 1, no. 4 (1983): 295-320.

Soloway, E. "Learning to Program = Learning to Construct Mechanisms and Explanations." *Communications of the ACM* 29, no. 9 (1986): 850-58.

Wing, Jeannette M. "Computational Thinking." *Communications of the ACM* 49, no. 3 (2006): 33-36.