Beyond tutorials: Using active learning to improve computational design instruction

Nick Senske[1]

[1]University of North Carolina at Charlotte, Charlotte, NC

ABSTRACT: This paper describes the development of an introductory computational design course for architects that uses active learning practices to improve student outcomes and engagement. The author presents results from a four-year impact study. Student performance improved with the introduction of active learning methods, specifically: peer learning labs, scaffolded design projects, and shorter lectures coupled with group discussion activities. In addition, students reported an improved perception of computing and an increased interest in the subject – a positive outcome for a required introductory course.

KEYWORDS: Education; Active Learning, Computational Design, Computing

1.0 INTRODUCTION

1.1. A required course for teaching computational thinking

In 2011, UNC Charlotte began integrating computational design throughout our curriculum. One of the core principles in this effort is the promotion of computational thinking in all courses and studios[1]. Computational thinking is an idea that originates from computer science, but has relevance to architectural design. To offer a simple definition: it is a mindset that helps one make the best use of humans and computers to solve complex problems (Wing 2006). Because this mindset is not limited to any particular tools or domains, we believe it can provide our students with a computing foundation that is both flexible and robust. Thus, to introduce our students to computational thinking, we teach a foundation-level course entitled *Computational Methods*.

A major premise underlying *Computational Methods* is that computational thinking is a form of literacy. This has two important implications. The first is that understanding computation and being able to express oneself computationally is not optional for architects anymore. Communicating and experiencing the world via computing is akin to reading and writing: it is a fundamental means of participating in today's profession and society in general. As such, *Computational Methods* is required for all our students and is a prerequisite for other computational courses (such as BIM, digital fabrication, etc.) later in the curriculum. The second implication of literacy is that computer programming is an essential skill. Using software is only reading it; a literate person must also be able to write (Kay 1993). Thus, learning to program (in some form) is also a requirement within our curriculum. This process begins in *Computational Methods*.

1.2. First iteration: traditional tutorials and lectures

The earliest version of the course (Fall 2011) devoted a majority of class time to tutorial-based labs. Students learned software (Processing and Grasshopper, in this instance) by watching the instructor and following along on computers in the lab. Later in the weekly lesson, the students attended a lecture, where the instructor summarized the week's concepts and discussed the connections between the tools and architectural theory and practice. The majority of assignments in the course were design-based. Students would be given some simple computational design parameters (e.g. create a visual composition using a nested loop) and asked to generate a design to demonstrate their understanding. A final, comprehensive project assessed students' ability to synthesize what they learned in the course. This pedagogy (or something similar) is common in many computing courses.

Unfortunately, the students' initial response to the course was below our expectations. In their semester evaluations, many complained that they found the pace and complexity of the material overwhelming. Because they struggled to make their scripts work, they could not come up with ideas for what to make. As a result, their design assignments suffered, as well. Even more concerning was that students did not find the course relevant to their professional development. They felt that learning programming was an unnecessary skill and they should be learning BIM instead. Despite our efforts to communicate the broad applications of computational thinking to their education, the students were unconvinced.

In response to these results, we set out to study the problem. Why was student comprehension low and why did so many fail to see the connections between the material and their work? Furthermore, what changes

could we make to our teaching methodology to improve student outcomes?

1.3. Confronting the challenges of teaching computational thinking

Although today's students spend much of their time using computers, this does not necessarily make them sophisticated users. Educational research indicates that learning computation presents a significant challenge for most people. In particular, learning to script can be difficult. Even expert computational designers admit as much (Burry 2011). But, in architecture, it can be hard to prove that an educational gap exists. Aside from Mark Burry's survey of practitioners in Scripting Cultures, we could find no empirical research on the problems architects face when learning computation.

Educational research on novices and non-majors learning to program shows that scripting is a challenging and complex cognitive activity (see a summary in Dalbey and Linn 1985). Learning a computer language is not the primary cause of these challenges, as many believe. Rather, the more difficult task is learning to think like a programmer (Winslow 1996). This appears to be a problem that applies to architects as well as those studying to be professional programmers. Our earlier research found that architecture students exhibit many of the same problems reading and writing programs that have been documented for novice computer science students (Senske 2014). Thus, how to teach computational thinking effectively is a problem that overlaps computer science and architecture, with many unanswered questions and opportunities for research.

1.4. Active learning

Introducing computation through traditional educational methods like lab tutorials and lectures may not effectively engage students to overcome these difficulties. Research suggests that instructors need to take into account differences in student learning styles and attention spans (Bransford et al. 2000). For most students, the passive consumption of content in school (e.g. how to use software) is not interesting. Worse, these lessons tend to impart brittle knowledge and skills which are quickly outdated. In architecture school, design projects are supposed to help students synthesize and critique what they study, but these may be inappropriate and counter-productive when students have only a superficial grasp of computation in the first place. Educational research suggests a different path towards helping students become better computational designers: active learning.

Active learning encourages students to play an active role in their own learning process: discussing, evaluating, and collaborating rather than merely listening and following along with instructions. It can help students develop a deeper understanding of complex subjects by engaging them in a peer-driven process of knowledge discovery, problem-solving, and critique (Prince 2004; Bonwell & Eison 1991). Instead of expecting students to make sense of computation by attempting to use it to design too early, this method is highly structured and promotes understanding by helping students become better learners. This is a more productive, sustainable goal than merely being a competent software user. The ability to learn computing better and apply it with a critical attitude can help students now and in the future as they adapt to technological and professional changes.

The remainder of this paper presents the introduction of active learning into Computational Methods and the results of a four-year study of its effectiveness. By sharing our rationale, course structure, and lessons learned, we hope to establish a model others can follow as they look for ways to effectively teach computation to a large and diverse audience.

2.0 BEYOND TUTORIALS

2.1. Second iteration, and beyond: active labs, scaffolded design, and active lectures

Computational Methods is a required foundation course for both graduates and undergraduates. It convenes in the fall term and meets twice a week for one 50 minute lab and one 75 minute lecture. The average course size is about 75 students, with a typical proportion of roughly 60% undergraduates to 40% graduate students.

The course syllabus is designed to introduce topics in a manner that carefully builds complexity and connections between each lesson. Many computing courses are structured around particular commands or tropes, which can lead to rote learning. In contrast, examples of *Computational Methods* topics include: abstraction, data structures, debugging, conditional logics, and problem solving with algorithms. These topics are not limited to a particular software implementation. Rather, they represent the "big ideas" of computational design. In addition, we frequently repeat concepts from week to week so that students have the opportunity to revisit them in different contexts. This is an application of the "spacing effect" in learning, which is said to improve comprehension (Dempster and Ferris 1990).

2.2. Flipped classrooms

In order to free up class time and resources to introduce active learning in the classroom, we restructured the way materials are presented to our students. Our tutorials and lectures are now available as online videos. This allows students to learn and review this material at their own pace, rather than the instructor's. With the additional time, we transformed our course schedule, teaching methods, and assignments to improve how students learn, practice, and form critical perspectives about the material. In practice, our teaching methodologies resemble a "flipped" classroom (Tucker 2012), where the students complete homework-like activities in class and watch videos of tutorials and lectures outside of class. Now that most of the content of the course is delivered outside of class, we use class time to synthesize, correct misunderstandings, and discuss the material.

2.3. Active labs

An example of one of these activities is our "active" lab, where students drive their own learning process. We use a form of peer instruction (Crouch and Mazur 2001) rather than the typical instructor-focused model. Before the lab, the students receive a tutorial video as homework. They watch the video to learn skills and concepts that will prepare them for the lab. When the students come to lab, instead of merely following the instructor (which would be a passive lab) they work with a partner on a self-directed "lab report". The report is similar to a science lab prompt, where the students are asked to construct, modify, and experiment with various computational systems (Figure 1). The students are presented with prompts to experiment with a system, answer questions about what they see, and use this information to understand the system and generalize conclusions.

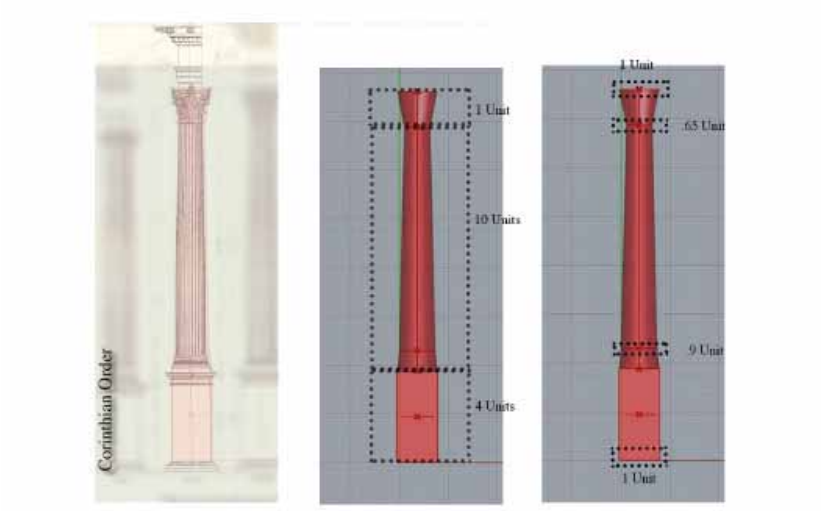

Figure 1: An example from an active lab report. In coordination with their history course, students diagram the proportions of Greek orders. These measurements form the basis of a column script that teaches the students how to create dependencies between geometric elements. The final script allows the students to study the orders while teaching them the power of parametric relationships. Source: (Author 2012)

2.4. Pair programming

Students work together in pairs on their lab reports, with one student scripting and the other guiding the process, frequently switching throughout the hour. This practice, known as pair programming (Figure 2), helps students manage the complexity of scripting and has been shown to improve both programming performance and enjoyment (McDowell et al. 2002; Nagappan et al. 2003). The instructor interacts with the pairs as a "coach", providing assistance and motivation where needed, but the students manage their own time in the lab. In general, the active lab requires students to engage in problem-solving and experimentation rather than merely following steps and learning software commands. The idea is for the students to ask and answer critical questions about how computational processes work. Our methods encourage students to develop computational thinking and allows instructors to more effectively assess that thinking.

Figure 2: Left: An “active” lab session featuring pair programming. Source: (Author 2014)

2.5. Scaffolded design projects

The lessons in *Computational Methods* feature less design than one might expect for an architecture course, but, based upon our experience, we believe that other forms of practice and assessment are more appropriate at this level. Design projects assume students can synthesize skills and ideas on their own, but may be inappropriate for novices who have an incomplete understanding of the material. Indeed, asking students to generate original computational designs too early may encourage rote thinking, as production does not necessarily require comprehension. Therefore, instead of traditional design projects, we assign “scaffolded” projects. Scaffolding is a practice wherein the instructor models a complex activity while providing pedagogical support for the students to approximate that activity (Collins et al. 1990). Over time, the support is removed and the student performs the activity on their own (hence, the scaffolding).

One example of a scaffolded project is a parametric precedent study (Figure 3). Instead of asking students to design something on their own parametrically, we guide them through a process where they reverse-engineer a complex form derived from a built project. In addition, the students learn to document and present a parametric process so that it is understandable to a non-computational audience. Scaffolding reduces the cognitive load for the students, so they do not have to recall so many commands or infer too much beyond what they know (Paas et al. 2003). They still have to investigate the precedents, experiment with solutions, and implement a final design – this is the “active” pedagogy at work – but the progression and goals are clearer than in many computational design prompts. Educational research suggests that, for novices, practicing knowledge is more beneficial than struggling to synthesize and generate knowledge (Collins et al. 1990). The use of scaffolded projects supports the critical inquiry we promote through the active labs, while providing students with the opportunity to express themselves

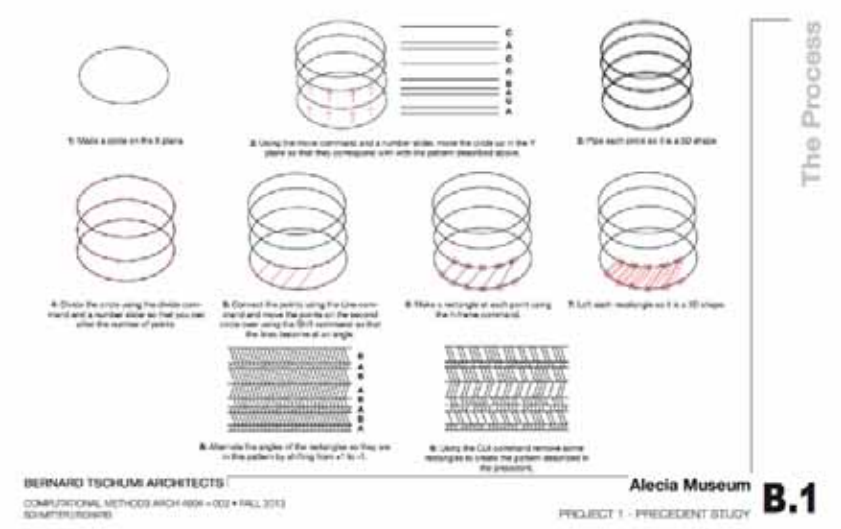

Figure 3. Assignments emphasize clear communication of process and computational ideas. Screen captures of scripts are not allowed. This image is an example of a student diagramming form generation within her parametric precedent. Source: (Author 2013)

2.6. Active learning classroom

The latest addition to the course, started in 2014, is a change from our traditional lectures to active learning sessions, facilitated by the introduction of the university's new active learning classroom (Figure 4). This new space features whiteboards, integrated laptops and monitors, and wireless microphones. Most importantly, instead of a lecture hall, with seats facing a podium, the space is filled with small group tables where the students face each other. The instructor gives brief, 5-10 minute lectures as a follow-up from the labs and videos earlier in the week. These talks are used to set up small group and class activities and discussions. For example, students might be asked to summarize concepts from a lecture on the whiteboard, diagram a script using touchscreens, or debate computational issues with the microphones. Similar to the active labs, the instructor walks around the room, encouraging discussions and clarifying points. A typical class period features several activities, which help the students reflect upon and develop their understanding of the week's topics.

We are still collecting data from this latest change to the course, but preliminary results suggest that students find the active classroom helps them understand computation better. At a minimum, the active learning discussions do not seem to have harmed student performance. We will report our findings shortly.

Figure 4: A group activity taught using the technology and specially-designed spaces of our active learning classroom. Source: (Author 2014)

In the next section, we will discuss how introducing active learning affected student performance and attitudes about *Computational Methods*.

3.0 METHODOLOGY AND ASSESSMENT

When *Computational Methods* first launched, we had no frame of reference for it within the curriculum and, therefore, no sense of how to measure its success. However, now that we have three years of data, we can identify trends and useful metrics[2]. If we compare these numbers from year to year, there is evidence that the active learning methods have helped *Computational Methods* perform better as an introductory course.

Table 1: Student Performance in Computational Methods. Traditional format in 2011; active learning format 2012-2014.

Year	# of students	Average score (%)	% Passing (students)
2011	71	83.1	87.5
2012	75	87.2	95
2013	74	88.5	97
2014	56	88.4	100

Our most critical metric is the pass/fail rate of the course. Because the course is required, we want to be sure our students keep up with the class and learn the material. In this sense, incorporating active learning appears to be an improvement. (Table 1) lists the pass rate of the course each semester, which we define as a C or better for both undergraduates and graduates. In the first iteration of the course, with traditional lectures and labs, the pass rate average between graduates and undergraduates was 87.5%. After updating the course pedagogy, the pass rate improved to 95% in 2012 and 97% in 2013. As of the last class (2014), the pass rate was 100%. Over this period, the course size has remained constant within statistical margins. By changing our teaching methods, without significantly altering the objectives or overall content of the course, more of our students pass the course than before[3].

In addition to examining objective course performance through assignments and testing, we developed a survey instrument to track student attitudes and impressions about the course. The surveys are online, voluntary, and anonymous and consist of basic demographic questions, a series of Likert-scale inventories, and short answer essays to account for answers outside the scope of the survey. Each year, we issue a pre-class and post-class survey, which we use to measure changes in our students over the semester. Our response rates are typically high, averaging close to 75% over the past three years.

The surveys have helped us measure our students' level of engagement. In other words: do they find the material relevant to their education and do they feel value in what we ask them to do? Positive affect is essential to promoting deep learning that transfers to other contexts (Pugh and Bergin 2006). If students do not think a course is worthwhile, they tend not to perform as well (Lepper 1985). In terms of affect, we found that the 2012 revisions made a difference. The most significant change in attitudes came in response to the post-class survey statement, "I think *Computational Methods* should be a required course." In 2011, only 36% of the class agreed. In fact, 26% strongly disagreed with the statement. The new instructional methods appear to have inverted this result. An average of the past two years' results revealed that 80% of our students now agree the course should be required and, of this cohort, 25% strongly agree. Students in the new version of *Computational Methods* not only believe the course is relevant, but they also report greater satisfaction from their participation in the course. In response to the statement "I am satisfied with my experience," 64.7% of Fall 2011 students agreed. This number improved to 94.2% and 98% in Fall 2012 and 2013, respectively. These evaluations are encouraging by themselves, but even more encouraging is the fact that they have remained consistent for the past two years. This suggests that they are the likely result of our new pedagogy and not a one-time occurrence.

4.0 REFLECTION AND FUTURE WORK

As of this writing, our assessment of *Computational Methods* is moving into a new phase. We are now conducting an in-depth analysis of past assignments to determine common misunderstandings and continuing to examine the impact of the course upon the rest of our curriculum through follow-up surveys. However, many questions are beyond the scope of our current studies. For instance, because we made a number of changes to the course between the first and second set of iterations, we cannot determine which of these interventions (or which combinations) resulted in improved student performance. A more controlled study could help clarify our findings. Another important question is how to more accurately measure computational thinking. We currently assess students' conceptual knowledge and success at specific tasks, but this may present an incomplete picture of their understanding. For instance, a student might provide the

correct answer to a question, but their process or reasoning may be flawed. In future research, we hope to improve our assessment techniques by observing how students apply computational thinking in problem-solving situations.

CONCLUSION

Introducing active learning into *Computational Methods* appears to be an improvement over traditional tutorial- and design-based pedagogies. Using empirically-proven teaching methods from other fields, we have improved how we teach computational design to a large audience with diverse backgrounds and aptitudes. Our students can competently apply computational skills and report a strong feeling of satisfaction with their efforts.

If a computationally literate profession is the goal, architectural education must change the way it teaches computing. *Computational Methods* proposes that a more learner-centered strategy can help architects learn the material better, but the pedagogy is still evolving and many key questions remain. We hope that by sharing our development process and results, our experiences may serve as a model for instructors and researchers to follow and improve.

REFERENCES

Bonwell, C. C., & Eison, J. A. 1991. Active Learning: Creating Excitement in the Classroom. 1991 ASHE-ERIC Higher Education Reports: ERIC Clearinghouse on Higher Education.

Bransford, J. D., Brown, A. L., & Cocking, R. R. 2000. How People Learn: Brain, Mind, Experience, and School. Washington, DC: National Academies Press.

Burry, M. 2011. Scripting Cultures: Architectural design and programming. New York: John Wiley & Sons.

Collins, A., Brown, J. S., & Holum, A. 1991. Cognitive Apprenticeship: making thinking visible. American educator, 15(3), 6-11.

Crouch, C. H., & Mazur, E. 2001. Peer instruction: Ten years of experience and results. American Journal of Physics, 69(9), 970-977.

Dalbey, J. and M. C. Linn. 1985. The Demands and Requirements of Computer Programming: a literature review. Journal of Educational Computing Research, 1(3): 253-274. Amityville, NY: Baywood Pub. Co.

Dempster, F. N., & Farris, R. 1990. The spacing effect: Research and practice. Journal of Research & Development in Education.

Lepper, M. R. 1985. Microcomputers in education: Motivational and social issues. American Psychologist 40(1): 1-18. Washington: American Psychological Association.

Kay, A. 1993. The Early History of Smalltalk. ACM SIGPLAN Notices 28(3): 69-95. New York: Association for Computing Machinery.

McDowell, C., Werner, L., Bullock, H., & Fernald, J. 2002. The effects of pair-programming on performance in an introductory programming course. In ACM SIGCSE Bulletin 34(1): 38-42. New York: Association for Computing Machinery.

Nagappan, N., L. Williams, et al. 2003. Improving the CS1 experience with pair programming. In SIGCSE Bulletin., 35(1), 359-362. New York: Association for Computing Machinery.

Paas, F., Renkl, A., & Sweller, J. 2003. Cognitive load theory and instructional design: Recent developments. Educational Psychologist, 38(1), 1-4.

Prince, M. 2004. Does active learning work? A review of the research. In Journal of Engineering Education, 93(3), 223-231.

Pugh, K. J., & Bergin, D. A. 2006. Motivational influences on transfer. Educational Psychologist, 41(3), 147-160. Washington: American Psychological Association.

Tucker, B. (2012). The Flipped Classroom. Education Next, 12(1), 82-83.

Senske, N. (2011). A Curriculum for Integrating Computational Thinking. In Parametricism: ACADIA Regional Conference Proceedings. Lincoln, NE., 91-98.

Senske, N. 2014. Confronting the Challenges of Computational Design Instruction. In Proceedings of the 19th International Conference of the Association of Computer- Aided Architectural Design Research in Asia - CAADRIA 2014, Kyoto, Japan, 831-842.

Wing, J. M. 2006. Computational Thinking. In Communications of the ACM, 49(3): 33-36. New York: Association for Computing Machinery.

Winslow, L. E. 1996. Programming pedagogy—a psychological overview. ACM SIGCSE Bulletin, 28(3): 17-22. New York: Association for Computing Machinery.