S. Chien, S. Choo, M. A. Schnabel, W. Nakapan, M. J. Kim, S. Roudavski (eds.), *Living Systems and Micro-Utopias: Towards Continuous Designing, Proceedings of the 21st International Conference of the Association for Computer-Aided Architectural Design Research in Asia CAADRIA 2016,* 829–838. © 2016, The Association for Computer-Aided Architectural Design Research in Asia (CAADRIA), Hong Kong.

'I'M A VISUAL THINKER.'

Rethinking algorithmic education for architectural design

MATTHEW AUSTIN and WAJDY QATTAN
University of Technology Sydney, Sydney, Australia.
matthew.austin@uts.edu.au
wajdy.qattan@student.uts.edu.au

Abstract. The representational and visual aspects of architectural design education cause certain pedagogical stresses in student's capacities to learn how to code, and this paper will serve as a critique of the current state of algorithmic pedagogy in architectural education. The paper will suggest that algorithmic curriculum should not frame code as 'a design tool', but as something to be designed in its own right; the writing of the code is the 'design brief' itself and not something additional to an architectural design brief. The paper will argue for an architecture-less educational environment that focuses on computational competencies such as logic, loops and lists along with building a strong analytical basis for students' understanding of programming and digital geometries.

Keywords. Pedagogy; algorithmic; programming; education.

1. Introduction

This paper will argue for a new mode of algorithmic education within the discipline of architecture in order to deal with new modes of architectural thinking. The rise of complexity theory and emergence has changed the way we frame and talk about architectural design in relationship between design outcomes and algorithmic processes[1]. Ideas such as 'top-down design' and 'bottom-up processes' have entered the way we frame and understand code within design processes. Although the introduction of these sorts of ideas into design is by no means new, it is becoming clearer that architecture lacks the educational frameworks in order to increase student literacy within the family of algorithms that embody these complex and emergent properties that seem to simulate the natural world.

The problem for architecture is two fold. Firstly, one would expect to see an increase in the complexity and sophistication of algorithms within architecture over time; however, this is not occurring. Instead there is an increase in the complexity and sophistication in understanding direct applications of the algorithm. Secondly, as algorithmic problems in architecture become more complex and dynamic, and thus the code required becomes more complex, new modes of algorithmic andragogy are required to bring students into a position where they can not only engage with these algorithms, but also innovate them. Further to this, as these problems become more complex, it is very likely that we require more algorithmic-literate architects to engage in these new modes of practice and problem solving in order for architecture to deal with complex and 'big data' problems.

This paper therefore positions itself as a critique on current algorithmic educational frameworks and attempts to question, understand and propose a solution to some of the deficiencies in current educational frameworks.

2. The nature of algorithmic education

Programming is undoubtedly difficult. Computer Science courses have one of the highest dropout rates of any course at a university level (Ljungkvist and Mozelius, 2012, p. 285) and many students have trouble learning even within introductory subjects (Lahtinen et al, 2005, p. 14). These subjects are not difficult because they are necessarily ill-structured or the tutelage is weak, but because the fundamental content is both theoretically and practically challenging to students, especially in fields in which students have no formal background (Eckerdal, 2009, p. 12; Robins et al, 2003, p. 138). Fundamental computer science ideas such as classes, functions, arrays, matrices, loops and gates are taught implicitly alongside their mathematical basis of number theory, set theory and logic.

Programming for architectural purposes adds the additional requirements of computational geometry, vector algebra and topology. To further conflate the problem, programming is often taught after a core competency with basic CAD, for example Rhino3D, is met. Students are therefore familiarized with user-friendly graphical user interfaces and thus programming within an integrated development environment is extremely alienating (Muller and Kidd, 2014, p. 176). Although Muller and Kidd (2014) are referring directly to teaching programming in a geography andragogy, they quite rightly argue that reintroducing mathematics, statistics and computer programming subjects at a later stage in student education can lead to anxiety, and architecture is no exception to this. This alienation and anxiety requires particular engagement from the educator in order to help students learning. Further to

this, material aimed at assisting students learning to code are generally ill-conceived and poorly designed. The Grasshopper Primer (ModeLab, 2015), is a particularly poignant example; its 242 pages rush through mathematical basics, logic and list structures at an alarming rate. Further, at no point does it attempt to meaningfully challenge the reader to engage with their learning, by coding something from scratch. Finally, to make the issue of programming andragogy worse, the educators within the field tend to have no formal computer science or mathematical education themselves. They are often self-taught making the process of framing core computer-science concepts difficult.

In short, a general algorithmic-design course in architecture is far more difficult than a programming fundamentals course. This is because more is taught faster and students in architecture have no fundamental understanding of computational geometry. For example, although students comfortable with any given CAD package can draw a B-Spline, very few know what it is or how it works. Although within the majority of architectural production, the technicalities of how geometry is computationally constructed is irrelevant, in the instance of programming understanding a geometric object's underlying construction is fundamental in framing how to build it from first principles and debugging potential problems that rise from this construction. This is not to say that we should change the way we teach CAD, as some students have no interest in programming, but that students lack the ability to understand the deficiencies in their knowledge (Kruger and Dunning, 1999, p. 30).

Further, this paper's second author has produced a series of interviews with computer science researchers revealing a general belief within the discipline that students require two to three programming courses in order to become sufficiently skilled (Nassir, 2015). It is important to note that in this instance 'skilled' does not refer to students capacity to know a 'programming language' but an understanding of programming fundamentals and logic (B. Faqeh, 2015). This begs the question: what has lead to the current curriculum expectation in architecture, that students with no deep knowledge of logic, programming or computational geometry can pick up algorithmic design and produce a 'good project' within twelve to fifteen weeks?

3. Unpicking the visual thinker

Architecture predominantly represents itself through visual communication, for example images, drawings and models. As such it is no surprise that many students, architects and academics consider themselves 'visual thinkers'. It is because of this visual nature of the discipline that architecture has

gravitated to more visual modes of programming, such as Processing and Grasshopper, in which designers are given visual feedback to help understand what a piece of code does.

However, visual programming has its limitations; it doesn't alleviate the lack of understanding about more non-visual fundamentals of programming and mathematics, for example, 4-dimensional transforms. Further to this, visual programming allows users to employ geometry without any understanding of its mathematical basis (Ozcan, Akarum, 2001, p. 27). Although at first glance this seems like an ideal situation in the andragogy of algorithmic design for architecture, it does make algorithms that are inherently object oriented, such as agent-based systems, far less obtainable to students learning to program. This is not to say that students would not be able to use an agent-based system, but that they would be far less likely to both understand it fully and write it from scratch.

Visual programming itself is not a problem as the majority of programming done within architecture is inherently visual. The issue is that most algorithmic design curricula within architecture rarely extend themselves outside the visual aspects of programming and reinforce these visual aspects through lesson and assessment design[2]. Most curricula direct students to create 'architecture' or at least something architectural as the goal of the course, rather than directly grade students on their capacity to program. Further to this issue, the fundamentals of programming are usually compressed into a smaller portion at the beginning of semester to make more room for design. This therefore transmutes the problem of 'learning to code' into a more comfortable 'architectural design problem'. This generally leads to a low percentage, somewhere between 5 to 15 percent, of students leaving an algorithmic design course knowing how to program at a basic level.

It is important to mention that architectural design andragogy is not merely problem solving but a creative endeavour in its own right, in which the outcome is judged by the merits of how the problem is solved and not just on the fact that it solves the problem (Harfield, 2007, p. 164). Further, this architectural design favours 'creative', 'novel' and/or 'good' outcomes not only from the perspective of the student themselves but also the tutor and coordinator. Although the relationship between outcome and process may be causal (i.e. a 'good' process implies a 'good' outcome), the process is generally only graded on its capacity to produce 'good' outcomes. Student focus therefore moves from learning the fundamentals of programming to 'designing through code'. Ironically, 'designing through code' is difficult, problematic and foolhardy without knowing the fundamentals of programming as students will know in a general sense what kind of outcome they want, however they lack the skills in order to produce it. This can likely lead to stu-

dents not using programming techniques and reverting back to methods of design they are personally comfortable with.

4. The Dunning-Kruger effect and algorithmic education

The Dunning-Kruger effect is a psychological phenomenon of cognitive bias where a relatively unskilled individual overestimates their abilities and their very lack of skills also robs them of the ability to realize they have over- estimated it (Kruger and Dunning, 1999, p. 30). For example, in the context of an architectural design studio, a failing or passing student may personally expect a grade of a credit, distinction or high distinction. This should come as no surprise to the reader. However, evidence suggests that the converse is also true, namely, that skilled individuals will underestimate their abilities (Kruger and Dunning, 1999, p. 33, p. 42-43). Using again the example of an architectural design studio a high distinction student may expect a grade of a distinction or a credit (while of course secretly hoping for an 'HD').

The Dunning-Kruger effect has been well studied and applied to understanding and grading student self-assessment within universities (Karatjas and Web, 2015, p. 30). What has not occurred, however, is a critique of curriculum design through this lens. Kruger and Dunning (1999, p. 43) argue that individuals that are skilled in a task, and thus find it easy, assume that other people would be equally skilled. This implies that individuals that easily learnt programming in architecture, which are the individuals most likely to be teaching programming in architecture, assume that other people will find it equally easy. This begs the question: Do algorithmic design curricula in architecture assume that students will retain fundamental programming concepts and logics beyond what is reasonable to expect? Considering the rate in which algorithmic design is taught, the amount of time that is spent within those curriculum on programming fundamentals in comparison to computer science courses, and the findings from Muller and Kidd (2014), it would not be foolhardy to suggest the answer to that question is *yes*.

It would, however, be foolhardy to suggest the solution to the problem is simply to go slower over a longer period of time. The problem, with algorithmic design in architecture, is that a lot of the simplest and most canonical algorithms (for example, voxelisation, recursive subdivision, packing, swarms, diffusion-limited aggregation, dynamic relaxation, cellular automata, etc.) require fairly well-established and fundamental knowledge of loops, logic gates, arrays, functions and classes in order for students to understand them, read them and write them. In other words the issue with algorithmic andragogy within architecture is that quite advanced programming concepts are required to produce quite basic visual outputs. This issue, along with the

Dunning-Kruger effect, therefore calls for a radical shift in the way that programming is taught within architecture schools.

5. The death of the tutorial

From the student's perspective, electives are a rare commodity. From the perspective of academia, core subjects are of equally rare value. Architecture requires knowledge of design, history, theory, construction, etc. Space in architectural studies is limited, and programming, in the scheme of a wider architectural education is a specialisation. Students are likely to only be interested in using one elective on programming, rather than the two to three subjects suggested by Nassir (2015).

The next problem is that it is likely that after completing an algorithmic design course, students will not program again. This may be because of subject focuses and tutor positions outside of the students' control, or a general dislike of programming. On this note, Bjork (2013, p. 2) states that the goal of instruction is to "equip the learner with the type of knowledge or skills that are durable and flexible". It is clear that current algorithmic andragogy in architecture struggles to retain 'durable' and 'flexible' knowledge within the majority of students. Nevertheless, if algorithmic design courses wish to educate more architecture students with base programming competencies, curricula need to be designed in a manner that focuses on the retention of programming fundamentals as its core goal.

Leonard (2015) quite aptly states that:

> Researchers have shown that, in laboratory tests, people quite consistently have "illusions" of competence. That is, they over-estimate their ability to solve future problems when they've been given a lot of help during lessons. When shown answers to questions, experiment subjects are likely to think they could have produced them ("Oh, sure, I knew that!") And the more familiar the material seems to them, the worse the students do in actually using it. Familiarity breeds complacency.

Leonard's statement implies a reasonable explanation for why students can understand programming concepts in particular contexts and not others. Showing students how to use abstract programming fundamentals in one context does not give 'flexibility' to the knowledge, and further it is only in hindsight that students see the solution to simple programming problems. Further, it allows the student to confuse performance, "…which is what can be observed and measured during instruction…" (Bjork, 2013, p. 2) with learning, that which is exhibited after instruction. This confusion between performance and learning suggests that the lab tutorial itself is a non-

productive mode of learning and should instead be replaced with, for lack of a better phrase, the lab problem. If we, for example, momentarily consider the famous turtle algorithm from LOGO in which users learn to move a turtle around a screen. Over time users would get better at moving a turtle around a screen and not better at programming algorithms as the problem is solved for them. Asking students to write code constitutes learning in a way that is relevant to them being able to solve different problems in the future.

Therefore algorithmic education should focus on setting students up to practice solving problems rather than following through a problem. In following through a problem students tend to copy the code without any understanding of its meaning. For example, students don't understand that they may have forgotten a '(' or a ':' in an IDE, or not add a number to a 'division component' in Grasshopper because there appears to be nothing to copy. If students regularly problem-solve while slowly understanding the abstract ideas of fundamental programming concepts, these ideas will remain abstract, flexible and more durable then standard tutorial methods. This mode of andragogy already exists within core architectural subjects; core design, construction and communication subjects already focus on practicing problem solving rather than following through problems. Further, it could be argued that this is intrinsic to architectural education. This is not to say that existing algorithmic design curricula do not practice problem solving, but that they practice solving architectural problems rather than solving programming problems.

What this suggests is that algorithmic design should be taught in a manner in which students are required to grapple with abstract programming fundamentals in a variety of different ways (Ekerdal, 2009, p. 11) in order to maximise learning at the cost of other significant architectural ideals such as 'good', 'novel' and 'creative' outcomes.

6. The architecture-less environment

At this point, it should be clear to the reader that the writing of an effective algorithmic design curriculum within architecture is a wicked problem. There is no clear solution, and any solution will have to sacrifice particular pieces of knowledge and trust students to develop those pieces of knowledge independently. This is under the assumption that the students understand why they should independently develop this knowledge and that they are engaged enough with their education to do so.

The authors propose that architecture itself should be thrown out of algorithmic design curricula, in order to shift the focus away from outcomes to a fundamental understanding of algorithmic processes. This is not to say that a

curriculum would remove students understanding of architectural applications of a said algorithm, but that it should not be directly assessed within the design of a curriculum. In other words, what if rather than grading the architectural potentials generated from an algorithm, the students understanding of the algorithm itself is graded? For example, the architectural potentials of cellular automata are ignored, and instead the focus is on the students writing Conway's Game of Life. The architectural applications and lack thereof of many canonical algorithms in architecture are already well known and can be set as a reading task at the beginning of a curriculum. Now students no longer focus on designing a 'good' outcome with cellular automata; students are free to research, program, and most likely struggle through, Conway's Game of Life, thus giving far more focus and understanding to core programming competencies, namely in this example, arrays, functions, loops, logic gates and classes. In short, the focus shifts from designing with code to designing code.

In context, a curriculum asking students to research and program a piece of code has several advantages over traditional modes of teaching algorithmic design in architecture. Firstly, the problem is not wicked like a design problem and as such it is very clear to the student what constitutes a 'good' outcome (i.e. a working piece of code) rather than a student's first foray into algorithmic architecture, in which both the outcomes and processes appear quite abstract. Secondly, the process of learning to program is stretched over a whole curriculum, and students are not disadvantaged for struggling to learn core skills quickly. Often algorithmic design curricula in architecture attempt to push towards 'good' design outcomes as quickly as possible, even though it is clear from Muller and Kidd (2014 p. 176), Eckerdal (2009), Robins et al (2003), Ljungkvist and Mozelius (2012, p. 285), and Nassir (2015) that not only some students will struggle with the content, thus being disadvantaged, but it is very likely that the majority of students will! And thirdly, it requires students to develop core-programming competencies along with analytic problem-solving skills. In other words, the goal of the curriculum's design is to give students the skills in order to program, which assumedly, is the reason that students would have enrolled in it. On this note, in this paper's first author's experiences, students can solve programming problems very easily if computers are not involved. For example, when students are given three hours to translate a given board game into pseudo code consisting of while loops, for loops and logic gates, the vast majority of students easily complete the task in the time limit. However, when students are asked to write a small piece of python script that draws a line between two points without assistance, many struggle and many simply give up. The point here is not that programming is difficult, but that this shift in curriculum fo-

cus doesn't give students the capacity to get through a curriculum via their architectural expertise, aesthetic sensibilities and visual-thinking skills, but pushes students to slowly solve logical-programming problems over the course of a curriculum rigorously and in detail.

This method of algorithmic andragogy falls short on other vital characteristics of a perfect algorithmic design course. It does not completely solve issues of student anxiety (Muller and Kidd, 2014 p. 176) and it doesn't solve the problem that students require additional subjects to complete their education on the topic. On a side note, it would appear wise that this method of algorithmic andragogy be considered a pre-requisite for algorithmic-design studio streams within core university subjects.

7. Conclusion

This paper's findings are in no way complete as to make any clear judgment about the success and failures of new modes of andragogy requires a substantial amount of research. However, this initial attempt highlights the issues with current modes of algorithmic andragogy within architectural education.

In conclusion, programming is undoubtedly difficult. Our current modes of algorithmic andragogy are severely limited by time, student capacity and curriculum design, and as such are unsatisfactory for successfully educating the majority of students. If architecture wants to deal with complex algorithms, emergence and 'big data', algorithmic architects need to move past visual programming and engage in core programming fundamentals. Albeit not perfect, the architecture-less environment offers several advantages over traditional modes of education and appears to have the capacity to produce a stronger programmer in conjunction with support from the core curriculum.

Endnotes

1. Although not all algorithmic processes are necessarily *digital*, in the sense that they use a computer, instances where students are required to write, read and understand code will be the focus of this paper.
2. This statement is made from the perspective of the authors. Having taught at several universities and seeing a wide range of differing algorithmic design core subjects and electives between the years 2009-2015 the general design of curriculum prioritising visuality has been a consistent theme.

References

Bjork, R.A.: 2015, "Learning verses performance". Available from: <bjorklab.psych.ucla.edu/pubs/Soderstrom_Bjork_Learning_versus_Performance.pdf> (accessed 4 November 2015).

Ekerdal, A.: 2009, *Novice Programming Students' Learning of Concepts and Practise,* PhD Thesis, Uppsala University.

Faqeh, B.: 2015, Interviewed by Wajdy Qattan, 16 August.

Harfield, S.: 2007, On Design 'Problematization': Theorising Differences in Designed Outcomes, *Design Studies*, **28**(2), 159–173.

Karatjas A.G. and Web J.A.: 2015, The Role of Gender in Grade Perception in Chemistry Courses, *Journal of College Science Teaching*, **45**(2), 30–35.

Kruger J. and Dunning D.: 1999, Unskilled and Unaware of It: How Difficulties in Recognizing One's Own Incompetence Lead to Inflated Self-Assessments, *Journal of Personality and Psychology*, **77**(6), 30–46.

Lahtinen, E, Ala-Mutka K., and Jarinen H-M.: 2005, A Study of the Difficulties of Novice Programmers, *ITiCSE 05 Proceedings of the 10th annual conference on innovation and technology in computer science education*, 14–18.

Leonard, D.: 2015, "Why Organizations need to Make Learning Hard", *Harvard Business* Review, Available from: < https://hbr.org/2015/11/why-organizations-need-to-make-learning-hard> (accessed 5 November 2015).

Ljungkvis P. and Mozelius, P.: 2012, Educational Games for Self Learning in Introductory Programming Courses – a Straightforward Design Approach with Progression Mechanisms, *Proceedings of the European Conference on Games Based Learning*, 285–293.

ModeLab: 2015, "The Grasshopper Primer V3.3", Available from: <grasshopper-primer.com/GrasshopperPrimer_V3-3-EN.pdf> (accessed 16 November 2015).

Muller, C.L. and Kidd, C.: 2014, Debbugging geographers: teaching programming to non-computer scientists, *Journal of Geography in Higher Education*, **38**(2), 175–192.

Nassir, S.: 2015, Inverviewed by Wajdy Qattan, 3 September.

Ozcan O. and Akarum L.: 2001, Mathematics and Design Education, *Design Issues*, **17**(3), 26–34.