Visualization Tools To Enhance College Level Introductory Programming Curricula.

Adam Lupu

Northwestern University

Wright (2010) calls for a "complete redesign" (p. 126) of college degree programs in the artificial sciences (a field including Computer Science, Information Systems, Software Engineering, and Information Technology). He claims that college programs fail to train students to become web programmers (Wright, 2010). Lending credence to this claim, Bennedsen and Caspersen (2007) found evidence that at least one third of students worldwide who enroll in a college-level introduction to programming course either fail or drop out. Such failure is unacceptable during a period when web programming is the most sought after artificial science skill in the jobs sector (Prabhakar, Litecky, and Arnett, 2005). I will examine one factor that contributes to this failure rate, namely the challenges students encountered while transitioning from natural language to computer language. Then I will show how two specific visualization tools (algorithm visualization technology and contextualized media computation) can be used to ease this transition, improve course pass rates, and legitimately prepare students for future employment as professional programmers.

People often describe tasks and procedures using "natural language" (Miller, 1981). For example, if I were to make a grocery list, then I might describe my decision making process using "if, then" statements. If I am out of milk, then add "milk" to the list. I will not necessarily include an "else" statement. As in, I will not be inclined to say, "If I am out of milk, then add 'milk' to the list, *else* do not add 'milk' to the list." To me, or to anyone I ask to make the grocery list, the "else" statement is implied. We omit such a statement in our everyday conversation, because it would sound unnatural. We also omit termination commands like "end." I would not tell myself, or my roommate, to stop making the list after it had grown to a certain size or after we had examined the refrigerator, freezer, and pantry for missing items. We would simply agree the list was "good enough." Miller (1981) found that when asked to describe the construction of a list, no one included “else” or “end” statements. Failure to include alternative instructions when conditions are not met or termination clauses when they are does not hinder our communication with other humans. It can, however, lead to several errors when communicating with a computer. To avoid these errors, students must adjust their natural communication methods when programming in languages that are legible to a computer (e.g. Java, C++, Python, etc.). According to Guzdial (2010) this adjustment requires students to think about tasks differently then they naturally would.

In addition to thinking differently about task descriptions, students must also deal with artificially constructed syntax and word use. For instance, to ask a computer to display a line of text using a common web programming language, Ruby, I may use the word “puts” followed by a colon (Puts: “grocery list.”). I would not naturally use this phrase to instruct a friend to write something down. The word’s natural meaning is loosely connected to my use for it, but its use in programming is not based on my natural language. This can create confusion for novice programmers because the meanings they associate with several words are no longer optimal when writing or reading a computer program. Students must learn a new meaning, which taxes their working memory and may cause them to lose sight of their programming objectives (Anderson, 1987). In addition to confusion over word use, there is potential for further confusion when teaching students how to use symbols and syntax. Miller (1981) found that humans tend to attribute meaning to symbols and words based on context. There are areas of research that focus on constructing context-specific computer languages, referred to as adaptive behavior languages (Simpkins, Bhat, Isabel, and Mateas, 2008). However, the leading languages in introductory computer science courses (Java, Python, and C++) are context-independent. Dehnadi and Bornat (2008) found evidence that students who assign inconsistent meanings and rules to unfamiliar symbols and syntax (context-dependent) perform far worse in introductory computer programming courses than those that apply consistent meanings and rules (context-independent). Therefore, introducing students to programming using context-independent languages can have a deleterious effect on comprehension and pass rates.

To examine one possible intervention to support students in using these languages, I would like to highlight the curricular use of algorithms. In many programming curricula, algorithms are introduced as sets of instructions written by a human user and read by a computer. Students find it challenging to learn to construct these algorithms while simultaneously learning a new computer language (Butler and Morgan, 2007). Students require guidance through the transition from natural language thinking to programming computer language algorithms. One powerful mechanism to facilitate this transition is the use of visualizations. Hundhausen, Douglas, and Stasko (2002), conducted a meta-study on the use of algorithmic visualization (AV) technology in introductory computer programming courses. AV technology includes animated films of algorithms in action, software that allows students to manipulate and create such films, and interactive programming environments for students to create their own visualizations. In the studies Hundhausen et al. (2002) reviewed, teachers did one of three things; they either 1) had students observe but not interact with animations, 2) engaged students in questioning, interacting with, or constructing algorithms using AV technology, or 3) did not use any form of AV technology (control group). Hundhausen et al. (2002) found student test performance improved when students interacted with AV technology, but only when they had to exhibit significant effort to do so. Passive observation of animations, whether during lecture or as assigned homework had an insignificant impact on student performance when compared against the control group (Hundhausen et al., 2002).

Hutchins' (1995) work on distributed cognition would seem to support these findings. He claimed that adding representational devices (AV technology in this case) extended cognition beyond the individual (the student) to include a system through which procedural memory propagates (Hutchins, 1995). In other words, rather than having two representations of an algorithm to draw from (natural language descriptions and computer language instructions) students now had three (natural language descriptions, computer language instructions, and visualizations). This may have resulted in a reduction on working memory demands, as students constructed visual animations or flow charts to hold and return (represent and transmit) information about an algorithm (Hutchins, 1995). Additionally, the active use of AV technology may have become "proceduralized." That is to say AV technology became a part of students' procedure for remembering, thinking about, and producing algorithms (Anderson, 1987, Hutchins, 1995). By contrast, passive observation of algorithm visualizations may not have prompted students to incorporate them into their "cognitive system" (Hutchins, 1995). Similarly, we might say that direct interaction with AV technology prompts students to "coordinate" the technology with lectures, readings, and homework (diSessa and Sherin, 1998). The addition of AV technology may have prompted students to "integrate" (include in the adjustment of their thinking/approaches/observations) a larger number of algorithm "aspects" (types of reading and producing), resulting in a more expert level of understanding and use (diSessa and Sherin, 1998).

AV technology's apparent value to improving student programming performance and expertise is notable given that AV technology is a subset of software visualizations. Software visualizations are used regularly in existing web programming jobs to help programming teams improve system performance, comprehend system structure and evolution, and track down bugs (Hundhausen et al., 2002). AV technology, therefore, has a direct relationship with the types of visualizations used by professionals in the web programming community. By actively using AV technology, students are participating in a professionally legitimate practice similar to the practices of their future employers and colleagues. Lave and Wenger's (1991) theory of legitimate peripheral participation (LPP) suggests this use of AV technology is a way to increase transfer from pedagogy to professional practice. In other words, a student who practices using AV technology in a classroom (peripheral to professional environments) would increase their likelihood of adapting to their first job, where they will engage in professionally legitimate practices using software visualizations. Therefore, not only does prompting students to actively construct/use algorithm visualizations help them perform better in introductory computer programming classes, it may also facilitate their transition into a community of professional programmers (Lave and Wenger, 1991).

AV technology is not the only visualization tool with pedagogical value. Bypassing the challenges with using artificial languages altogether, Guzdial (2010) developed an introductory computer-programming curriculum that replaces computer languages (Java, Python, C++ etc.) with a plethora of contextualized media. Instead of trying to learn programming while simultaneous learning a computer language, students using Guzdial's curriculum learn programming tasks through cleverly constructed media manipulation exercises. For example, students learn to iterate through the elements of an array by converting all the pixels in a picture to grayscale or by removing red-eye from a picture without altering the red pixels elsewhere. Instead of concatenating (joining) strings of code, they concatenate sound buffers or video filters while splicing digital media (Guzdial, 2010). By asking students to perform operations on media that are similar to the operations found in software built by professional programmers, Guzdial's curriculum prompts students to engage in legitimate practices (Lave and Wenger, 1991).

Additionally, Guzdial's curriculum uses visualization tools to guide students through programming tasks without requiring them to transition from natural language to computer code.

Lou and Fletcher (2009) refer to this abstraction of programming beyond computer language as computational thinking (CT) and argue in favor of its use in computer science curricula. "In the absence of [language-based] programming, teaching CT should focus on establishing vocabularies and symbols that can be used to annotate and describe computation and abstraction, suggest information and execution, and provide notation around which mental models of processes can be built." (Lu and Fletcher, 2009, p. 260) Guzdial's contextualized media curriculum challenges students to annotate and describe the computations they observe in modern software programs (red-eye reduction, video editing, photo manipulation, etc.) It prompts them to analyze these computations, execute them, and annotate their processes (Forte and Guzdial, 2004). Since its introduction, contextualized Media Computation has contributed to the improvement of student pass rates at Georgia Tech, Gainesville State College, University of Illinois-Chicago (UIC), and University of California, San Diego (UCSD) (Guzdial, 2010).

But will the skills students pick up in a Media Computation class transfer when they are eventually required to use computer languages? The fact that the exercises in a Media Computation curriculum are situated in existing professional programming practices suggest that transfer is more likely to occur (Lave and Wenger, 1991). Since the media manipulation exercises use the same "conditions" as language based programming (e.g. iterating on elements of an array, joining modification instructions) and require the same "productions" (e.g. systematic grayscaling, applying multiple video filters) the likelihood of transfer increases (Anderson, 1987). In fact, even though the curriculum is still relatively new, there is some evidence to suggest students can successfully transfer the computational thinking they acquire through contextualized media to language-based programming. UCSD students who took the introductory class (CS1) using the Media Computation curriculum also performed better than their peers (those who took a traditional CS1 class) in the language-based second level computer programming course: CS2. (Simon et al., 2010)

To summarize, students struggle with the transition from natural language to computer language when learning to program. This is one suggested reason for the poor performance and inadequate pass rates in introductory college programming courses. Informed and deliberate application of visualization tools can help ease students through this transition. Specifically, the use of algorithm visualization technology and contextualized media computation also benefit students by preparing them to participate in professional programming communities after college.

References

Anderson, J. R. (1987). Skill acquisition: Compilation of weak-method problem situations. *Psychological Review*, *94*(2), 192-210. doi:10.1037/0033-295X.94.2.192

Bennedsen, J., & Caspersen, M. E. (2007). Failure rates in introductory programming. *SIGCSE Bull.*, *39*(2), 32–36. doi:10.1145/1272848.1272879

Bornat, R., Dehnadi, S., & Simon. (2008). Mental models, consistency and programming aptitude. *Proceedings of the tenth conference on Australasian computing education - Volume 78*, ACE ’08 (pp. 53–61). Darlinghurst, Australia, Australia: Australian Computer Society, Inc. Retrieved from http://dl.acm.org/citation.cfm?id=1379249.1379253

diSessa, A. A., & Sherin, B. L. (1998). What changes in conceptual change? *International Journal of Science Education*, *20*, 1155-1191. doi:10.1080/0950069980201002

Hutchins, E. (1995). How a cockpit remembers its speeds. *Cognitive Science*, *19*(3), 265-288. doi:10.1016/0364-0213(95)90020-9

Forte, A., & Guzdial, M. (2004). Computers for communication, not calculation: Media as a motivation and context for learning. *Hawaii International Conference on System Sciences* (Vol. 4, p. 40096a). Los Alamitos, CA, USA: IEEE Computer Society. doi:http://doi.ieeecomputersociety.org/10.1109/HICSS.2004.1265259

Hundhausen, C. D., Douglas, S. A., & Stasko, J. T. (2002). A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages & Computing*, *13*(3), 259-290. doi:10.1006/jvlc.2002.0237

Lave, J., & Wenger, E. (1991). *Situated learning: legitimate peripheral participation*. Cambridge University Press.

Lu, J. J., & Fletcher, G. H. L. (2009). Thinking about computational thinking. *SIGCSE Bull.*, *41*(1), 260–264. doi:10.1145/1539024.1508959

Miller, L. A. (1981). Natural language programming: Styles, strategies, and contrasts. *IBM Systems Journal*, *20*(2), 184-215. doi:10.1147/sj.202.0184

Guzdial, M. (2007). *Why is it so hard to learn to program?* Making Software: What Really Works, and Why We Believe It, ch. 7, O’Reilly Media, Inc.

Prabhakar, B., Litecky, C. R., & Arnett, K. (2005). IT skills in a tough job market. *Commun. ACM*, *48*(10), 91–94. doi:10.1145/1089107.1089110

Simon, B., Kinnunen, P., Porter, L., & Zazkis, D. (2010). Experience report: CS1 for majors with media computation. *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education*, ITiCSE '10 (pp. 214–218). New York, NY, USA: ACM. doi:10.1145/1822090.1822151

Simpkins, C., Bhat, S., Isbell,Jr., C., & Mateas, M. (2008). Towards adaptive programming: Integrating reinforcement learning into a programming language. *SIGPLAN Not.*, *43*(10), 603–614. doi:10.1145/1449955.1449811

Wright, K. (2010). Capstone programming courses considered harmful. *Commun. ACM*, *53*(4), 124–127. doi:10.1145/1721654.1721689