AC 2009-10: DISTANCE LEARNING AND COGNITIVE LOAD THEORY TO IMPROVE TRADITIONAL AND NON-TRADITIONAL STUDENT LEARNING OF COMPUTER PROGRAMMING FOR MECHANICAL ENGINEERS: QUANTITATIVE ASSESSMENT

Thomas Impelluso, San Diego State University

Dr. Impelluso received his BA in Liberal Arts from Columbia University. This was followed by two MS degrees in Civil Engineering and Biomechanics, also from Columbia. He received his doctorate in Computational Mechanics from the University of California, San Diego. Following this, he worked for three years in the software industry, writing code for seismic data acquisition, visualization, and analysis. He then commenced post-doctoral studies at UCSD, wherein he secured grants in physics-based virtual reality. He is now a tenured associate professor at San Diego State University, revisiting and researching human bone remodeling algorithms and muscle models using advanced tools of the cyberinfrastructure. He has created a curriculum in which students learn mechanics not by using commercial simulation software, but by creating their own. His interests include opera, sociology, and philosophy. He is currently enjoying teaching his two young children how to ride bicycles.

Distance Learning and Cognitive Load Theory to Improve Traditional and Non-Traditional Student Learning of Computer Programming for Mechanical Engineers: Quantitative Assessment

ABSTRACT

This paper reports on the re-design of a computer programming class for students of mechanical engineering. The content was re-designed using Cognitive Load Theory; the delivery was re-designed using on-line technologies. Student learning was objectively assessed; it improved and the drop-out rate reduced. A previous paper reported on greatly improved student attitudes and instructor reviews. This paper reports on objective data: comparing student performance on identical final exams. Note is made of improved learning by non-traditional engineering students. This paper also reports on two additional teaching strategies that were deployed to improve learning. Finally, this work points to the next step in this evolving redesign.

Introduction

Cognitive Load Theory (CLT) provides guidelines to present information in a manner that encourages learning and optimizes intellectual performance [1]. As an example, consider the obstacles in learning new material in a non-native language. Clearly, there is an overload: learners must master both the new material and the language itself. Interestingly, this is resonant with the challenge of learning to program a computer which faces those students not in the computer science major. Such learners must master both extraneous issues such as the operating system and the compiler and then the intrinsic issues such as the syntax of the language and application areas. CLT can mitigate challenges in such cases when learning loads are diverse and high.

According to CLT, information can be stored in long term memory after first being properly integrated, by working memory, into a mental structure that represents the schema of the material. However, the faculty of working memory has limits and this, unfortunately, can hinder learning, especially when many extraneous and ancillary facts compete to challenge the cognitive learning loads (which, in the case of programming, encompasses text editing, operating systems and compilers). CLT posits that there are three basic types of cognitive loads placed on a learner:

- "Intrinsic cognitive load" was first described in 1991 [2] as the essential material to be learned. Accordingly, all instruction has an inherent difficulty associated with it and this intrinsic material may not be altered by an instructor. In learning a foreign language, this includes the vocabulary and syntax.

- "Extraneous cognitive load" is generated by the manner in which information is presented to learners [3] and, in the case of a programming language (and, specifically here: for non-computer scientists), the ancillary information regards text editors, compilers, and operating systems. (In the case of a spoken language, information is presented using technologies that must be mastered, such as laboratories with recorders.)
- "Germane cognitive load" was first described by Sweller, van Merrienboer, and Paas in 1998 [4]. It is that load devoted to the processing, construction, and automation of schemata necessary to integrate knowledge into consciousness. This includes motivations to learn and how the knowledge is conveyed in the rest of the curriculum such as reading novels, or programming mathematical algorithms.

These three loads are additive in the learning process and research suggests [4] that when courses are redesigned with due respect paid to the interaction of cognitive loads, learning is improved. For example, while intrinsic load is thought to be immutable, instructional designers can exercise the option to manipulate extraneous and germane loads. With complex material, it is best to strive to minimize the extraneous cognitive load and maximize the germane load.

Table 1
The Three Types of Cognitive Loads Placed on Learners of Computer Programming

Load	**Examples**
Intrinsic	Syntax: data types, loops, logical tests, arrays, functions
Extraneous	Ancillary tools including: text editor, operating system and the compiler
Germane	Numerical algorithms in computational mechanics

Table 1 presents this author's view of the various learning loads experienced in computer programming. The intrinsic learning load is generally high in computer programming – it involves logic and syntax. Thus, if one also employs methods that add an extraneous load (such as complex compiler interfaces), it is very likely that there will be little capacity left for germane load that might be used to motivate students of mechanical engineering to learn programming; and the ensuing overload would then hinder their learning.

Furthermore, other research has shown that complete, thorough, and fully commented programming examples provide greater motivation for novices than simply working out problems from scratch [5][6]. Although this may seem counterintuitive, tests have demonstrated that studying complete examples facilitates learning more effectively than actually solving the equivalent problems [7]. Additionally, in many cases, a variation of worked examples, balanced with assignments, has been used and studied [8]. Students can be urged to *complete* the solution, which is only possible with the careful study of the partial example provided in the completion task. As with providing completely worked examples, this serves to decrease extraneous cognitive load [9].

In this course redesign, distance technology was used to minimize extraneous load. Scaffolding was used to enhance the germane load. Scaffolding is an instructional method wherein layered material is presented to the student and then removed later as students develop their own learning strategies. Scaffolding was an essential ingredient in this CLT-based redesign, and this approach is supported by existing research that has been successfully applied to the domain of computer programming [10].

2. Course prior to redesign

2.1 Course content

The course under discussion is SDSU: ME203—Programming. In this class, the C programming language is taught in a UNIX environment. The course presents a *procedure oriented* language (as opposed to *object oriented* language such as Java or C++), because mechanical engineers are more concerned with the *process* of applied mathematical algorithms (solids, fluids, thermal studies) than with *objects* to be manipulated (computer graphics, bioinformatics). Of the procedure oriented languages, C was selected because it is the language in which most operating systems are written, as are network, math and graphics libraries.

Thus, the focus of the class was on the syntax of the C language. However, advanced syntax techniques such as "data structures" were also taught. Secondary attention was paid to the Gauss Reduction method and various algorithms to multiply matrices: the exclusive focus was syntax. Mechanical engineering coding examples were not integrated into the course; they were presented without instructional design forethought.

2.2 Course delivery

Prior to Fall 2006, the class met physically and the exclusive method of content delivery was through face-to-face lecture. Instruction was provided in a workstation laboratory. This laboratory was a dedicated computational resource cluster of 30 UltraSPARC models 170 and 170E workstations using the Sun Grid Engine software from Sun Microsystems. Each station in the cluster had 128MB of physical memory, and contained one 167MHz US-I CPU. The workstations were interconnected using high-speed network infrastructure from Myricom.

The instructor taught at one workstation and displayed his monitor on an overhead projector. Students were able to watch the instructor discuss the code line-by-line, compile it, and run it. Then, students would work on their own code in separate lab sessions. This model of instruction had weaknesses. First, the size of the class was limited to the number of workstations. Furthermore, the workstations had to be upgraded every few years at considerable expense. Third, the students often expressed frustration as to why they were learning the material – the course was not taught specifically for mechanical engineers. Student reviews consistently mentioned that there was no reason for mechanical engineers to learn programming. Thus, course redesign was initiated.

3. Course after redesign

3.1 Course content

Two levels of course material were scaffolded by themselves and with each other: (1) the syntax of the language, and (2) the applied mathematical algorithms (vector, matrix manipulation, Gauss Reduction and Newton-Raphson methods). The goal was to avoid previous student criticisms of seeing no purpose to learning programming. The scaffolding more tightly connected the syntax to the algorithm and gave motivation for mechanical engineering students.

3.1.1 Vertical Scaffolding

The left column of Table 2 provides the *syntax structures* that were discussed in the redesigned class. Complete and commented code syntax examples were scaffolded on the skeleton of preceding ones: loops were discussed in the context of logical structures, and arrays were discussed in the context of loops. As the students progressed through the material, their understanding of vector manipulation, became essential to subsequent material. The right column indicates the *mathematical algorithms* that were discussed in class: Complete and commented algorithms were scaffolded on previous ones. Thus, Newton-Raphson relied on the Gauss Reduction code; Gauss-Reduction relied on matrix manipulation, and matrix manipulation relied on vectors. The same code "grew" and "evolved" in each example and with deliberation and consistency.

Table 2
The Scaffolding of Algorithm and Syntax After Redesign of the Course

Syntax	Algorithm
Data types and logic	Temperature conversion
Logic & loop formality	Bisection method
Logic & loop formality	Newton's method
Logic & loop formality	Numerical integration
Input/Output	Repeat of all algorithms
Arrays	Matrix-vector manipulation
Arrays, files	Gauss reduction
Arrays, files, functions	Gauss reduction with functions
Arrays, files, functions, memory	Newton Raphson method

3.1.2 Horizontal Scaffolding

The driving focus of the content in this class was the algorithm rather than the syntax. Thus, this inverts the way programming has traditionally been taught, in which syntax rules are presented and are the focus—as often happens when programming classes are farmed out to computer science departments in which either the theory of coding, or operating systems, is more the focus. This in no way is intended to disparage those departments; rather, it is to indicate that such approaches to programming may not be suitable for mechanical engineers. Furthermore, there were no coding examples of sorting, alphabetizing, or interest rate problems that plague introductory computer programming courses for mechanical engineers.

Table 2 presents the algorithms in the first column, followed by the programming syntax in the second. Vertically, the table demonstrates the order in which both topics were addressed. It is important to note that numerical algorithmic convergence and stability issues were ignored, in lieu of the most simple algorithm implementation. This table also demonstrates the horizontal scaffolding that occurred in the class. By connecting algorithm to construct, the redesign invigorated the germane load of the student learners.

Initially, the instructional goal was to challenge the students to read codes as if they were a new language. It was not expected that the students master the code's nuances and reproduce them at this stage. Rather, the goal was to immerse the students in a new language and expect them to follow the general idea of how the language implements the logic of a games and algorithms. In this way, a syntax construct was linked to an algorithm construct. As the student mastered each level, it was subsumed into the subsequent ones.

After that initial introduction, the course progressed into a series of code examples involving matrix manipulations. Next, it moved on to the two core concepts of the course. In fact, students were often reminded that these two algorithms were the core algorithms in mechanics. The Gauss Reduction is one algorithm used to solve a system of linear equations, while Newton-Raphson is implemented for a system of non-linear equations; both are critical components in mechanics analysis. However, there is serendipitously something more profound which was exploited here: The Newton-Raphson method builds upon the Gauss Reduction method. This creates an overarching structure to the class as it drives toward the study of very simple non-linear systems: everything was built upon the previous codes.

3.1.3 Temporal Scaffolding and Motivation

There was, however, a third type of scaffolding. Students of mechanical engineering traditionally learned programming to implement the algorithms (Gauss Reduction, Newton-Raphson) that are critical to their sub-disciplines (Finite Element Methods, Multi-Body Dynamics Methods, Computational Fluid Mechanics). There are a wealth of sites on the internet where one can view mpgs of these algorithms functioning. Thus, each class session in this redesigned course began by viewing one of these films. In this way, students were introduced to how programming is critical to the discipline of mechanical engineering.

.

The author refers to this as "temporal scaffolding" and it was used to introduce has motivational pieces were used in the course: during the roll-out of the course, students experience professional programs that implemented each algorithm and syntax they encountered. In this process, each day of the class is scaffolded back to the first day, and forward to the last. In this way, the class takes on an integral architecture in which students realize that every aspect of programming and every algorithm they encounter is not only related to each other, but to what precedes it and what follows it; and also to their discipline.

3.2 Course Delivery

With regard to delivery, two modes were used in equal parts: (1) face-to-face, and (2) interactive, online desktop sharing. Half the classes were face-to-face, and this is when and where the algorithm and syntax were taught. Extensive PowerPoint slides were developed and they were tied to each item in Table 2. Face-to-face lectures focused on the interplay of intrinsic and germane learning loads.

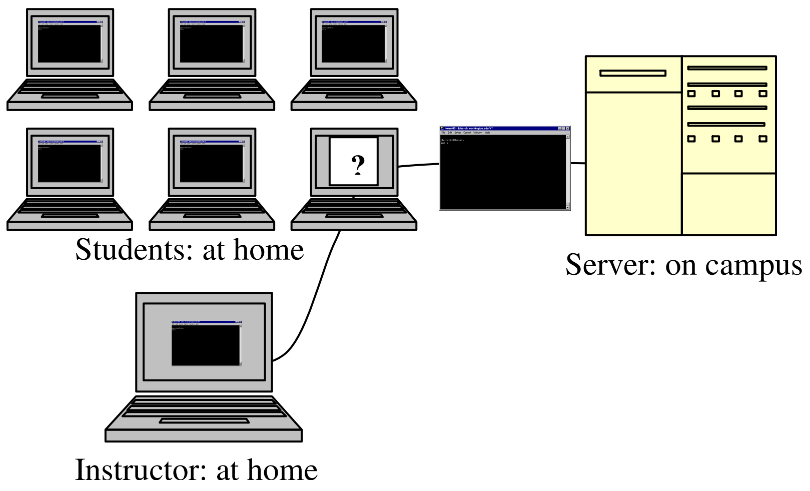

Figure 1. Student-instructor connections for online class sessions.

Horizon Wimba was the method used to conduct on-line, synchronous instruction. However, this was not a passive use in which students simply observed lectures over the internet. Application sharing technology was used—the instructor took control of student laptops as if working with the student, side-by-side, while also demonstrating the effort to the rest of the class. The schematic for an online session is indicated Figure 1. Students were able to work on their assignments from home (during or after class lab-time), regardless of their operating system, by first establishing a network connection through a secure shell (SSH) to the server on campus using a monitor prompt. Once connected to the server, students are able to write, compile, and debug their codes.

The instructor also maintained an SSH connection – again, a simple terminal window which enabled remote logins – to the same server and exploited the application sharing interface of the instructional des. The instructor shared his desktop (which contained an SSH connection to the server) with the class and demonstrated the process of writing, compiling, debugging, and running example codes. Occasionally, whether during a class session or during on-line office hours, a particular student would request assistance with an assignment (indicated by the laptop with the "?" mark and his SSH connection by the largest black monitor window external to his laptop). At those times, the instructor activated the Wimba application sharing interface and asked the student to share his or her desktop with him and the rest of the class. Then, the instructor addressed the student's questions, while also sharing the information with all of the students. And, of course, the session was recorded and archived for all to play at their leisure.

All the students in the class used the same operating system on which to compile and run their code examples—this added a layer of homogeneity to the instruction and this was reassuring to the students. The students used an SSH tool to connect to the common server on which they all studied and learned. This statement warrants focused reiteration. All the writing and compiling occurred in a common workstation environment, unencumbered by the nuances of diverse compilers. This consistency—this common computational environment—reduced the extraneous load of learning operating systems, compilers, and text editors; students were able to focus attention on the syntax of the language and the mathematical algorithms.

4. Student Learning Outcomes: Objective Student Performance Data

This course redesign was subjected to two levels of assessment. Student attitudinal assessment was conducted first. Students were extremely receptive to the redesign: both content and delivery. Surveys indicated that this excited the students and motivated them to learn. However, this data has been reported elsewhere [11]. Furthermore, as a result of this redesign, it had also been reported that the class size was increased—there was no longer a need for a physical workstation laboratory—the computer laboratory became virtual. Thus, the enrollment was able to triple and this reduced instruction cost. This brings us now to objective assessment.

Successful course redesign should, in principle, result in higher instructor reviews by students and improved grades. The risk, of course, is that instructors could be accused of grade inflation to secure improved student attitudes. For example, the designer/instructor of this class knew, from the outset, that student learning and student attitudes were improving and that if he gave a final exam of consistent caliber to that given before the redesign, the grades would improve. So, to avoid the risk of the accusation that the improved attitudes resulted from grade inflation, the instructor kept making the final exam increasingly difficult during the four semesters of this redesign. It was for this period that student evaluations of the instructor were assessed—so clearly, the reviews were not based on easy grading. This has also been reported elsewhere [11].

Once student anecdotal evidence had been gathered, and the course redesign had undergone two iterations, the instructor gave the same final exam in Fall 2008 that he had given in Fall 2006 (before the redesign)—realizing that this might lead to grade inflation, but taking the risk. Table 2 presents the final exam grades for Fall 2006 (column 2) and Fall 2008 (column 3). Row 1 presents the average final exam grade for all students who registered for the class. Row 2 removes from the average those students who scored a perfect 0.0 on the final (i.e., they did not take the exam, or dropped the class). Such data is repeated in row 3 and 4 for Hispanic students: row 3 for all Hispanic students, and row 4 for all Hispanic students who also received a perfect 0.0 on the final exam. This data is repeated again in rows 5 and 6 for female students: row 5 for all female students, and row 6 for all female students who received a 0.0 on the final exam. The parenthetical in all cells represents the sample number.

Table 3
Comparative Objective Student Performance on Identical Final Exams

Row	Category of Student	Fall 2006	Fall 2008
1	All students	39.44 (27)	66.41 (85)
2	All students who did not score a 0.0	53.25 (20)	76.28 (74)
3	All Hispanic students	30.62 (8)	69.16 (25)
4	All Hispanic students who did not score a 0.0	61.25 (4)	82.33 (21)
5	All female students	35.00 (4)	88.25 (5)
6	All female students who did not score a 0.0	70.00 (2)	88.25 (5)

The author reiterates that the final exam and the instructor were the same for both classes. Overall, the average grade increased as a result of the course redesign: For those students who took the final exam, grades increased by 43.24%. In addition, before the redesign, a drop-out rate of 25.9% (as occurred in Fall 2006) was typical. After the redesign, the drop-out rate decreased to 13%.

The performance of Hispanic students was noticeably improved as a result of this course redesign: For students who took the final exam, scores increased by 34.41%. However, the data tells an additional story; the dropout rate changed dramatically. Before the redesign, 50% of the Hispanic students failed to take the final exam, while after the redesign, only 16% dropped out. Thus, this group also performed better and their drop-out rate was greatly reduced.

Although the numbers were smaller, there is still data regarding female student performance. Female student performance was also noticeably improved as a result of this course redesign: For female students who took the final exam, average scores increased by 26.07%. Here too, a closer inspection of the data in the table reveals even more. Prior to the redesign, the instructor was losing half the female students. After the redesign, all female students finished the course. Thus, for this group too, the overall performance increased and their drop-out rate vanished.

When coupled with the previous research [11] and the current paper, it is clear that this course redesign has invigorated the learning process for students. It is now incumbent upon the author to engage in a more formal student questionnaire to assess what specific aspect of the redesign led to improved learning, reduction in drop out rate, and greater student excitement (perhaps the solitude of distance learning is more appealing to non-traditional students). The next phase in this redesign should reveal this.

5. Discussion and Conclusions

The on-line learning helped reduced the extraneous load while, at the same time, saving money on the up-keep of a laboratory. By using desktop sharing, the instructor was able to guide students through nuances extraneous details – operating system, compiler, text editor – while sharing with all eight students, simultaneously. Rather than help one student by a workstation, the instructor was able to help all. At the same time, this also reduced infrastructure cost.

5.1 Online for a Purpose

SDSU has a policy that a course is "hybrid" if 45% of instruction occurs online. This compelled the course designer to initially hold physical sessions (55%) and online sessions (45%) indiscriminately and one after the other. In the second roll-out, however, the instructor began using the online sessions for laboratories to diminish extraneous load. The author advises adopters of distance learning to seek out which aspects of a course are suitable for distance learning and deploy those with the technology. Distance learning can be a powerful tool when used with purpose and not simply as another vehicle to deliver content; and, in this case, delivery was guided by CLT as a pedagogical theory.

As a result of this experiment with distance learning, student learning improved dramatically. But there is more: The workstation cluster once used to instruct the class in exclusive face-to-face lectures is no longer needed. This has already amounted to nearly $40,000 in savings to the department. Furthermore, the distance delivery has enabled the class to overcome the enrollment limitation dictated by the available workstations. Enrollment has progressed from under 30 students to now over ninety in one session, obviating the need to hire a lecturer to support additional sections. As of this writing, the original laboratory has been removed and the space is now utilized for other classrooms.

5.2 Surveys

Lacking focused student assessment surveys at this time, the author can only surmise a the reason for the success with regard to non-traditional students of engineering, based upon informal conversations. The online modules enable a modicum of privacy in the learning process and undermines the sense of an "old-boy" network in learning computer programming for non-computer scientists. This will be specifically addressed in surveys of next semester's students.

5.3 Final Thoughts

As the initial Captivate sessions rolled out in conjunction with Wimba, physical attendance began to drop. It has occurred to the instructor that he is on the way toward a model of instruction wherein faculty become tutors guiding the advanced students and spending focused time guiding students having difficulty. In due time, judicious blending of interactive and passive online modules, coupled with proper scaffolding of material, will result in a class attended only by those having difficulty. More advanced students will pace themselves and the instructor will be able to extend focused instruction to students facing difficulty in the learning process.

The purpose of this course redesign was to improve student learning, and also to demonstrate new reasons for mechanical engineering students to embrace computer programming. Today's software, such as Matlab, is relied upon to teach the algorithms of mechanics. Unfortunately, such point-and-click environments with established interfaces add a layer of obfuscation around the simplicity of studying mechanics algorithms. Despite this, departments are now opting out of teaching computer programming in favor of such math packages. Perhaps the diversity of operating systems and compiler issues drives this motion. This paper (and its companion paper on attitudinal data [11]) demonstrates that it is possible to teach computer programming to mechanical engineers in a way that encourages their learning.

Today's machines do not simply think, they also communicate with each other. Further, some mechanical engineers of tomorrow need to know the language of how machines communicate. In addition, games and film are now adapting physics modules for more realistic animation, so again, it is in the interest of mechanicians to involve themselves with this discipline, too. Programming is an essential skill, not only for machines and simulation science, but to understand the fundamentals of the algorithms that undergird the discipline of mechanical engineering. This course redesign re-developed a traditional course in programming and the result was improved student learning and excitement about the discipline of mechanical engineering by itself, and as it intersects other, emerging disciplines.

REFERENCES

[1] Sweller, J. 1994. Cognitive load theory, learning difficulty and instructional design. *Learning and Instruction* 4: 295–312.

[2] Kalyuga, S., P. Chandler, and J. Sweller.. 1998. Levels of expertise and instructional design. *Human Factors* 40: 1–17.

[3] Pollock, E., P. Chandler, and J. Sweller. 2002. Assimilating complex information. *Learning and Instruction.* 12, no.1: 61–86.

[4] Sweller, J., J. Van Merriënboer, and F. Paas. 1998. Cognitive architecture and instructional design. *Educational Psychology Review* 10 no.3: 251–296.

[5] Renkl, A., and R. Atkinson. 2003. Structure the transition from example study to problem solving in cognitive skill acquisition: A cognitive load perspective. *Educational Psychologist* 38 no. 1: 15–22.

[6] Renkl, A., R. Atkinson, and C. Grosse. 2004. How fading worked solution steps works: A cognitive load perspective. *Instructional Science*: 59–82.

[7] Van Gerven, P., F. Paas, J. Van Merriënboer, and H. Schmidt. 2002. Cognitive load theory and aging: Effects of worked examples on training efficiency. *Learning and Instruction* 12: 87–105.

[8] Van Merriënboer, J., and F. Paas. 1989. Automation and schema acquisition in learning elementary programming: Implications for the design of practice. *Computers in Human Behavior* 6: 273–289.

[9] Van Merriënboer, J., J. Schuurman, M. de Croock, and F. Paas. 2002. Redirecting learners' attention during training: Effects on cognitive load, transfer test performance, and training efficiency. *Learning and Instruction* 12: 11–37.

[10] Van Merriënboer, J. 1990. Strategies for programming instruction in high school: Program completion vs. program generation. *Journal of Educational Computing Research* 6 no.3: 265.

[12] Lighter, S., M. J. Bober; and C. Willi. 2007. Team-based activities to promote engaged learning. *College Teaching* 55: 5–18.