Breadth-First CS 1 for Scientists

Zachary Dodds, Christine Alvarado, Geoff Kuenning, Ran Libeskind-Hadas
Harvey Mudd College Computer Science Department
301 Platt Boulevard
Claremont, CA 91711
909-607-1813

{dodds, alvarado, geoff, hadas}@cs.hmc.edu

ABSTRACT

This paper describes an introductory CS course designed to provide future scientists with a one-semester overview of the discipline. The course takes a breadth-first approach that leverages its students' interest and experience in science, mathematics, and engineering. In contrast to many other styles of CS 1, this course does not presume that its students will study more computer science, but it does seek to prepare them should they choose to do so. In addition to describing the curriculum and resources, we summarize our preliminary assessments of this course and a comparison with the more traditional, imperative-first introduction it replaced. The data thus far suggest that this *CS for Scientists* course improves our students' understanding of CS, its applications, and practice.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer Science Education

General Terms

Measurement, Design, Human Factors

Keywords

CS for scientists, introductory CS, CS 1 assessment

1. CS FOR SCIENTISTS

Scrutiny seems an unavoidable fate for introductory computer science. In a field as dynamic as CS, we who teach CS0 and CS1 should strive to remain relevant and current. At the same time, we try to retain those topics and skills that enable our students to cope with next year's changes as well as last year's. This balance is particularly delicate when designing introductory CS for scientists. The evolving impact of CS on all scientific disciplines has been dramatic and well documented, e.g., [27][29]. As George Johnson put it, "All science is computer science." [17]

In contrast, many scientific programs of study present facets of CS only as needed: programming skills and styles may derive from a particular language or environment, e.g., Matlab or LabView. This approach presumably keeps such programs relevant and up-to-date, but it emphasizes particular tools over the broader computational skills so vital in all areas of science today.

To leverage CS's growing importance, we replaced our traditional CS 1 course with a breadth-first version *nicknamed CS for Scientists* in 2006 [1]. Our goal was to create a curriculum "suitable for any student intending to major in science or engineering (including CS students)." [24] In particular, we hoped this new offering would (1) develop programming and problem-solving skills useful across engineering, mathematics, and the natural sciences, (2) attract students to continue studying CS, and (3) provide a coherent, intellectually compelling picture of computer science, even as final CS course.

1.1 Context and Related Work

It is a wonderful time to teach CS 1! Curricular innovations within introductory CS are both inspiring and numerous. Many of these experiments draw strength in a similar manner: by weaving a *thematic structure* amid introductory CS topics [22][15].

One of the most widespread of these themes for introductory CS is media computation [14][20]. Other themes now scaffolding CS1 include games [3][13][18][30], robotics [5], computer vision [21], and art [12][25]. In each of these cases, the thematic overlay tends to pull away from CS and toward the specifics of the course's theme. Throughout *CS for Scientists* we strove to keep the focus on CS, with applications motivating that focus.

Science and engineering enjoys a long history as a CS theme [2][16][19][26]. Yet these experiments, both new and old, tend to be service courses rather than CS *per se*, e.g., they do not contribute to a CS degree. Courses like [8] and [28] present facets of CS to specialists in other disciplines. Our course, on the other hand, represents a full-fledged CS 1 designed to generate interest in and prepare students for additional courses within the field. Although we feel our students' ability and work ethic are unusual, we also believe that scaled versions of this course could serve the computational requirements of future scientists from a wide variety of backgrounds.

Pedagogically popular styles of CS 1, such as imperative-first or objects-first [10], all make the implicit assumption that there will be something *second*. We knew that only a fraction of our students would continue with CS, though we hoped to make it a sizeable fraction! We hypothesized that breadth-first would best suit students for whom the class might also be *breadth-last*.

ITiCSE'07, June 23–27, 2007, Dundee, Scotland, United Kingdom.

Breadth-first CS is far from new. As CC2001 summarizes aptly,

> the breadth-first model has not enjoyed the success that its proponents had envisioned… most breadth-first courses that exist today seem to be lead-ins to a more traditional programming sequence. This model, which has several successful implementations, is outlined in CS100B… [10]

Our *CS for Scientists* takes this hard-won experience to heart; CS100B is our curriculum's basis. Yet our course has significant shifts in emphasis to serve future scientists and engineers: multi-paradigm programming, leveraging existing code, CS's influence on science today, and acknowledging the reality that many students would not be able to take another CS course afterwards.

Perhaps the work closest to *CS for Scientists* is Sedgewick and Wayne's forthcoming text [24] and the Princeton University course, COS126, from which it has grown. While our debt to COS126 is strong in spirit, several differences distinguish our offering. First, rather than work only within the object-oriented paradigm offered by Java, we take a multiparadigm approach via Python. Because of this change, we have contributed a freely-available body of Python software to support our assignments and laboratories. Further, we have been able to compare students' responses to different programming styles and evaluate our curriculum and students' gains from it.

1.2 Contributions

Thus, this paper presents four specific contributions that *CS for Scientists* adds to the effort begun in [24]:

- *A complete CS 1 curriculum*. Two fifteen-week sets of homework assignments and lecture slides are online: one set for students with no programming experience and another for students with a year (or more) of programming background.
- *Guided laboratory exercises*. Accompanying the assignments and slides are step-by-step exercises used during structured laboratory sessions each week through the semester.
- *Support software*. We have written and tested supporting Python code for simulating a simple computer architecture, its machine and assembly languages, easily accessible 2d graphics, and manipulating audio files.
- *Evaluation*. All of the materials above have been classroom-tested. We report initial results from weekly assessments of student engagement and understanding and we compare the new curriculum with the prior Java-based CS 1 course.

The data thus far suggest that CS for Scientists succeeds in three ways: students "get" the importance of CS to their future scientific endeavors; students also feel they can leverage their CS skills to support those endeavors. Students are attracted to continue studying CS at the same rate as from traditional alternatives. It is our hope that the curriculum, support materials, and assessments presented here will be of use to other CS educators serving future scientists, mathematicians, and engineers.

2. CURRICULUM

186 first-year college students took *CS for Scientists* in the fall of 2006: 74 of them in an accelerated section for those with at least a year of high-school programming experience and 112 of them in a section for those with no CS or programming background. All 186 students are pursuing a degree in some natural science, i.e., mathematics, engineering, physics, chemistry, biology, CS, or combinations of these fields. Though they had acknowledged an interest in science, the majority of the students have not yet chosen their major field of study, a decision not required until the second year at our institution.

To implement our breadth-first curriculum, we broke the semester into five three-week units (see Table 1) in which students would learn and practice different programming paradigms. In order to support all of these with a minimum of syntactic overhead, we heed [6] and [9] in choosing a multi-paradigm language, Python.

Table 1. Summary of *CS for Scientists'* Curriculum

Weeks	Paradigm	Samples of the labs and assignments
1-3	functional	integration, random walks, ciphers
4-6	low-level	recursion in assembly, 4-bit multiplier
7-9	imperative	Markov text generation, game of life
10-12	objects/classes	Connect Four player, sudoku solver
13-15	CS theory	uncomputability, finite-state machines

2.1 Modules, example labs, and assignments

To underscore the role of small, task-specific functions as the basic building blocks of computation, we started with functional programming, i.e., functional (de)composition, map, reduce, higher-order functions, and lambda expressions. While software engineers will persuasively argue that objects and classes are computation's basic building blocks, the programming needed most often by practicing scientists and engineers is smaller in scale, e.g., quick scripts to analyze, summarize, or reformat data. Moreover, by starting with a functional approach we built upon students' prior ability and comfort with mathematical functions.

The second module, "low-level" computation, reinforced the idea of modularity and composition via logic gates: students built 4-bit ripple-carry adders from AND, NOT, and OR gates in Carl Burch's outstanding Logisim tool [7]. Those adders then became building blocks in 4-bit multipliers. Combinational-circuit design segued to larger-scale computer architecture: students capped this experience by implementing recursion (the stack and function calls) in **Hmmm**, a Python-based assembly language simulator.

The transition is smooth from the register-level `jmp` and `cmp` assembly instructions to the repetitive control structures and variable reassignment of imperative programming, our third module. We felt that students with high-school Java experience needed supplemental material: implementing Huffman coding and Mastermind kept them engaged in what was otherwise largely review. In labs all students implemented Mandelbrot-set drawing and John Conway's Game of Life using vPython [23].

Students then augmented procedural programming with class- and object-based constructs. They created a `Date` class to answer questions like "how many days apart are June 25, 2007 and February 30, 1712" and "which day of the week is most likely to be the 13th of the month."† Implementing Connect Four and a

† Sweden observed Feb. 30, 1712, which fell 107,852 days before June 25, 2007. Friday is strictly more common as the 13th than any other day of the week.

sudoku solver provided deeper design practice with both object-oriented programming and 2d arrays.

A medium-sized final project further exercised object-oriented style, with students choosing among three options: a physical simulator and GUI for a game of pool; a state-based controller that could navigate a simulated robot through its environment; or a finite-automaton simulator, with graphics, of a small space-filling agent similar to Karel [4].

The finite-state machines used in the latter two final projects complemented in-class exercises on (un)computability and deterministic finite automata. These lectures, reinforced by final-exam questions, wrapped up the term with a bird's-eye view of what computation can and cannot do. Examples included the halting problem, several other uncomputable functions, and an abstract perspective on program state and models of computation.

Figure 1 shows examples of different students' work from fall 2006 – all from the novice programmers' section. They highlight some of the different examples of interfaces used throughout the course. Details on the support software follow.

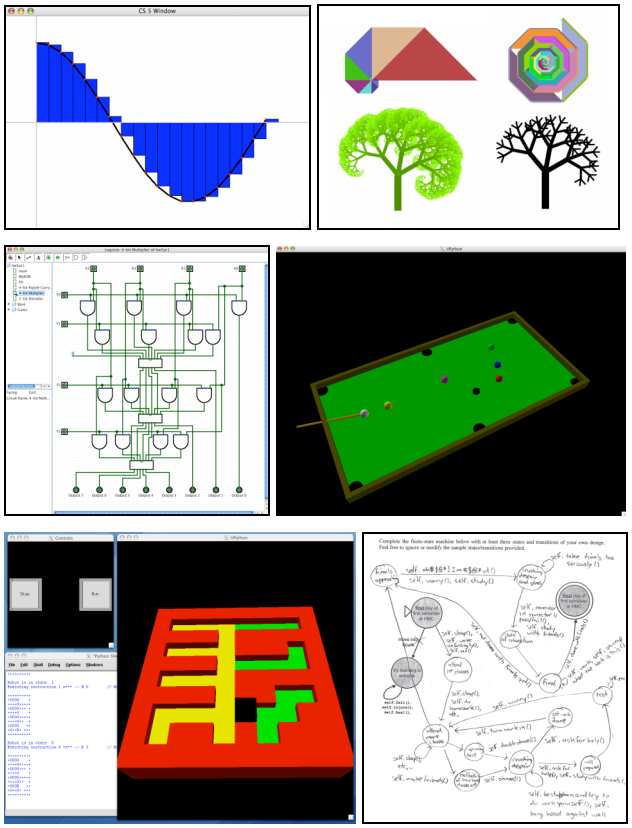


Figure 1. Student work from fall 2006's *CS for Scientists* ***Top left***: visualizing numeric integration ***Top right***: turtle-graphics art ***Middle left***: a four-bit multiplier circuit ***Middle right***: physics-based pool, a final project ***Bottom left***: a Karel-like automaton exploring its environment ***Bottom right***: a finite-state machine submitted on the final exam (featuring object-oriented syntax!)

Lecture Slides The course used seventy 50-minute Powerpoint presentations (30-40 slides each) that constitute the foundation of the course materials. Without a text, we sought to use visual representations as much as possible; less than a quarter of the slides are text-only. Figure 2 shows two slides from the second module on circuits and assembly language programming.

Labs, Assignments, Exams We used a set of guided 2-3 hour laboratories, one per week, and two sets of 4-6 hour out-of-class assignments – one for the accelerated and one for the beginner's section of the course. On average, students completed four programming problems per week. Students took two 1-hour midterms and a 3-hour final.

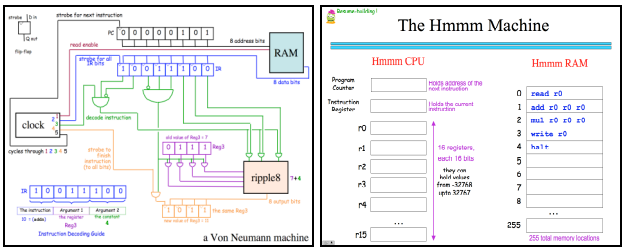


Figure 2. Course slides on Hmmm, module 2's assembly language, and a bit of potential circuit-level implementation.

Support Software To supplement the freely available Logisim [7] for circuit design and vPython [23] for 3d graphics, we have created several Python-based tools for *CS for Scientists*:

- The **Hmmm** assembler and simulator. **Hmmm** is a 16-instruction, 8-bit assembly language for a 256-word machine. It has a Python assembler and simulator/debugger. Simple enough for CS1, **Hmmm** is powerful enough to enable small recursive implementations: students built recursive factorial and towers-of-Hanoi programs in it. The slides in figure 2 diagram the **Hmmm** machine and a bit of its (potential) circuit-level design.
- A **`csplot.py`** package for 2d graphics. In contrast to John Zelle's excellent **`graphics.py`** package [32] that introduces students to writing their own interfaces, **`csplot`** provides autoscaling and mouse-based translation and zoom. **`csturtle`** extends this package with a Logo-like turtle interface; Python's builtin **`turtle.py`** lacks rescaling and recentering. These tools are *visualizers* for computational results more than resources for learning GUI programming. The top left and right images in Figure 1 show **`csplot`** and **`csturtle`**, respectively
- A **`csaudio.py`** package that enables reading, manipulation, and writing of sound files. While Mark Guzdial's JES and MediaTools [14] incorporate audio processing into a unified IDE using Java, this lightweight package is a stand-alone Python implementation that runs on PCs and Macs alike.

Indeed, all of the support material for the course runs on Linux, Mac OS X, and Windows. Portability was important: our labs are Mac-based; many students worked on their own PCs running a Microsoft OS; a few diehards used their own Linux installs, too. The course does not use a text; hence, all of these materials come from the course's website [1].

3. EXPERIENCES AND EVALUATION

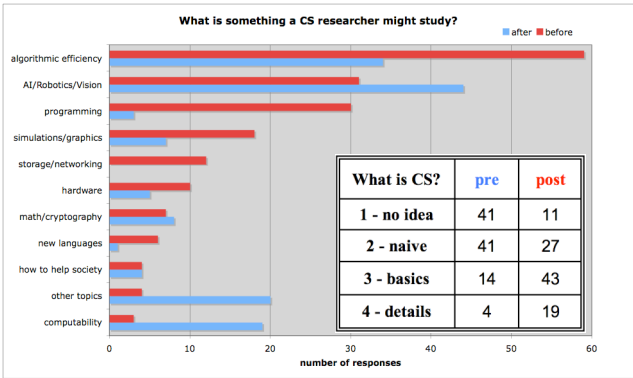
Although not all of our institutional course-evaluation data is available to us as of this writing, we have assessed the impact of the course in the following four ways:

3.1 Pre- and post-term surveys

To assess the course's success in conveying both the import and span of CS, students completed pre- and post-term surveys asking "What is computer science?" and "Describe one thing a researcher in CS might study."

The responses to "What is CS?" have been coded into four levels of sophistication: **Level 1 (none)** represents non-answers such as "the science of computers" or "the study of technology," as well as purely derivative/analytic ones, e.g., "using code to get computers to do things" or "figuring out how computers work." Responses that articulate some of synthesis or breadth within CS are **Level 2 (naive)**, e.g., "software and hardware design" and "coding, debugging, and analyzing problems to develop computer-based solutions." Answers that acknowledged the field beyond physical computers and their software became **Level 3 (basics)**, e.g., "the study of computational algorithms and their applications." Finally, the most nuanced answers become **Level 4 (details)**: "a lot is about general, language-agnostic even system-agnostic algorithms and relative merits of speed and efficiency and in some cases … actually wondering how and if it is possible. CS is not programming, it is implemented in programming."

The pre- and post-survey percentages appear in the inset, below. The bar chart summarizes the most commonly cited "things a CS researcher might study." Note that the latter categories were not supplied, but the result of a clustering of the responses provided. HCI and UI design appeared a good deal in the "other topics" list after the course, but not at all beforehand.



What is CS?	pre	post
1 - no idea	41	11
2 - naive	41	27
3 - basics	14	43
4 - details	4	19

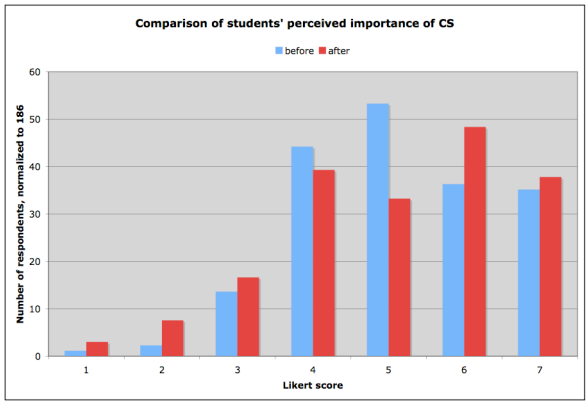
We were both surprised and heartened by this assessment: surprised by many of the impressions of CS articulated by our incoming undergraduate science students, but heartened by the substantial differences evident by the end of the semester.

In the initial surveys, many students clearly saw CS as a derivative field whose practitioners investigate artifacts created by other scientists: a typical comment was "[a CS researcher would] study what makes computers work." By the end of the term, this perception had completely disappeared. We also feel the substantial drop in the answer "programming" confirmed our success at conveying the breadth of CS beyond the skills that superficially characterize the discipline in many students' minds.

3.2 Affective assessment of CS's impact

We measured students' perceptions of the impact CS will have on them personally by asking "how important you think CS or programming skills will be in your future" with Likert-scale responses ranging from 1 – "not at all" to 7 – "very."

While the means of these before-and-after distributions are identical at 5.1, their shapes are significantly different. Indeed, the heavier tails on both ends of the scale suggest that more students have "taken sides" as to whether or not they feel comfortable and eager to draw upon and build upon their CS skills in the future.



3.3 Comparison with a traditional CS 1

Because this course replaced a procedural-then-objects Java course, similar in approach to [11] and [31], we were interested in comparing the responses of the two courses' students to the statements ***(A) The course stimulated my interest in the subject matter*** and ***(B) I learned a great deal in this course***. The table below shows a significant uptick in student agreement with these statements – unsurprising, perhaps, given that *CS for Scientists* was designed to better address its students' interests. Even so, we feel these data do support both the approach and curriculum. The first and last rows show how CS1 compared with *all* of our college's courses in all departments.

Table 2. Comparing student agreement with (A) and (B)

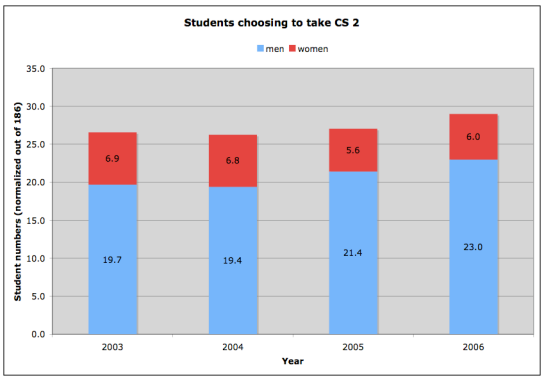
Students Considered	Agreement with (A), from 1-7	Agreement with (B), from 1-7
Fall 2005: all courses	5.65	5.71
Fall 2005: traditional CS1	5.14	5.81
Fall 2006: *CS for Scientists*	5.89	6.11
Fall 2006: all courses	5.70	5.80

3.4 Subsequent CS Enrollment

A full comparison between *CS for Scientists* and the more traditional CS1 course it replaced will not be possible until formal course evaluations get back to the instructors. However, an initial assessment of comparative student impact can be based on the numbers of students who choose to enroll in CS2 in the spring semester following the two versions of CS 1.

The figure above shows the trend from the last 4 years, including a breakdown by student gender. Continuing enrollments are normalized to this year's CS1 class size of 186. Increasing the number of students *majoring* in CS was not a goal of the redesign, though we had hoped for an increase. Even as additional, more thorough investigations into these numbers are underway, we are

happy that this change has at least maintained the numbers both of students and of women who choose to continue studying CS.



4. VERDICT

From these data and in looking back broadly at our initial offering of *CS for Scientists*, we are optimistic about its approach to teaching future scientists introductory computer science.

As always, there remain a number of rough edges that we look forward to addressing next fall. Some assignments (Connect Four) required more time than we wanted students to spend; others frustrated students because the tools (vPython) were as new to us as them. Additional final-project options would have been welcome, and we would also like to schedule homework-based reinforcement of module 5, computability and state machines.

More generally, we intend to pull more examples from mathematics, engineering, physics, chemistry, and biology to strengthen the overarching theme of the class. This first offering has convinced us that enabling students to choose their own path through a set of lab and homework problems alleviates the differences in background inevitable in any large class. Further, multiple pathways permit students to personalize the course content without sacrificing our central focus on CS itself.

Overall, we feel our experience with *CS for Scientists* provides additional evidence of the effectiveness of *thematically structured* introductory courses. We look forward to working with other CS educators targeting math, science, and engineering students to benefit from and improve upon the curriculum, software, and insights gained from this experiment.

5. REFERENCES

[1] CS 5 website, https://www.cs.hmc.edu/twiki/bin/view/CS5/WebHome

[2] Bachnak, R. and Steidley, C. An interdisciplinary laboratory for computer science and engineering technology. *Journal of Computing Sciences in Colleges* 17(5) April 2002, 186-192.

[3] Bayliss, J. D. and Strout, S. Games as a "flavor" of CS1. In *Proc. SIGCSE 2006; Houston, TX, USA*[A], 500-504.

[4] Bergin, J., Roberts, J., Pattis, R., and Stehlik, M. *Karel++: A Gentle Introduction to the Art of Object-Oriented Programming*. John Wiley & Sons, NY, NY, 1996

[5] Blank, D. Robots Make Computer Science Personal. *Communications of the ACM* 49(12) (Dec. 2006), 25-27.

[6] Budd, T. A. and Pandey, R. K. Never mind the paradigm, what about multiparadigm languages? *ACM SIGCSE Bulletin* 27(2) (June 1995), 25-30.

[7] Burch, C. Logisim: a graphical system for logic circuit design and simulation. *Journal on Ed. Resources in Computing (JERIC)* [A] 2(1) (3/2002), 5-16.

[8] Burhans, D. T. and Skuse, G. R. The role of computer science in undergraduate bioinformatics education. In *Proc. SIGCSE 2004; Norfolk, VA, USA*[A], 417-421.

[9] Close, R., Kopec, D., and Aman, J. CS1: perspectives on programming languages and the breadth-first approach. In *Proc. CCSCNE 2000; Mahwah, NJ, USA*[A], 228-234.

[10] Computing Curricula 2001. *Journal on Educational Resources in Computing (JERIC)* [A], Joint Task Force on Computing Curricula, eds. Volume 1, Issue 3es (Fall 2001).

[11] Crescenzi, P., Loreti, M. and Pugliese, R. Assessing CS1 java skills: a three-year experience. In *Proc. ITiCSE 2006; Bologna, Italy*[A], 348.

[12] Davis, T. A. and Kundert-Gibbs, J. The role of computer science in digital production arts. In *Proc. ITiCSE 2006; Bologna, Italy*[A], 73-77.

[13] Giguette, R. The Crawfish and the Aztec treasure maze: adventures in data structures. *ACM SIGCSE Bulletin* 34(4) (Dec. 2002), 89-93.

[14] Guzdial, M. A media computation course for non-majors. In *Proc. ITiCSE '03; Thessaloniki, Greece*[A], 104-108.

[15] Guzdial, M. and Tew, A. E. Imagineering inauthentic legitimate peripheral participation: an instructional design approach for motivating computing education. In *Proc. ICER 2006; Canterbury, UK*[A], 51-58.

[16] Jehn, L. A., Rine, D. C., and Sondak, N. Computer science and engineering education: Current trends, new dimensions and related professional programs. In *Proc. SIGCSE 1978; Pittsburgh, PA, USA*[A], 162-178.

[17] Johnson, George. All Science is Computer Science. *New York Times* March 25, 2001.

[18] Ladd, B. C. The curse of Monkey Island: holding the attention of students weaned on computer games. *Journal of Computing Sciences in Colleges* 21(6) (June 2006), 162-174.

[19] Lambrix, P. and Kamkar, M. Computer science as an integrated part of engineering education. In *Proc. ITICSE 1998; Dublin, Ireland*[A], 153-156.

[20] Matzko, S. and Davis, T. A. Teaching CS1 with graphics and C. In *Proc. ITiCSE 2006; Bologna, Italy*[A], 168-172.

[21] Olson, C. F. Encouraging the development of undergraduate researchers in computer vision. In *Proc. ITiCSE 2006; Bologna, Italy*[A], 255-259.

[22] Paul, J. Leveraging students' knowledge: introducing CS 1 concepts. *Journal of Computing Sciences in Colleges* 22(1) (Oct. 2006), 246-252.

[23] Scherer, D., Dubois, P., and Sherwood, B. VPython: 3D interactive scientific graphics for students. *Computing in Science and Eng.* 2(5) 2000, 56-62.

[24] Sedgewick, R. and Wayne, K. *Introduction to Programming (in Java)*, preliminary version, Pearson Addison Wesley, 2006. ISBN 0-536-31807-7.

[25] Smith King, L. A. and Barr, J. Computer science for the artist. In *Proc. SIGCSE 1997; San Jose, CA, USA*[A], 150-153.

[26] Stevenson, D. E. Science, computational science, and computer science: at a crossroads. In *Proc. ACM '93; Indianapolis, IN, USA*[A], 7-14.

[27] Steering the future of Computing. *Nature* 440(7083) (March 2006 special issue on 2020 Computing), 383-580.

[28] Tesser, H., Al-Haddad, H. and Anderson, G. Instrumentation: a multi-science integrated sequence. In *Proc. SIGCSE 2000; Austin, TX, USA*[A], 232-236.

[29] Towards 2020 Science, by the 2020 Science Expert Group. Microsoft Press, Redmond, WA, USA. 2006.

[30] Wallace, S. A. and Nierman, A. Addressing the need for a java based game curriculum. *Journal of Computing Sciences in Colleges* 22(2) 12/2006, 20-26.

[31] Weir, G. R. S., Vilner, T., Mendes, A. J., and Nordström, M. Difficulties teaching Java in CS1 and how we aim to solve them. In *Proc. ITiCSE '05; Caparica, Portugal*[A], 344-345.

[32] Zelle, J. *Python Programming: An Introduction to Computer Science*. Franklin, Beedle & Associates. Wilsonville, OR. 2004. ISBN 1-887902-99-6.

[A] ACM Press, New York, New York, USA