Design Process for a Contextualized Non-majors Computing Course

Mark Guzdial
College of Computing/GVU
Georgia Institute of Technology
801 Atlantic Drive
Atlanta, Georgia
guzdial@cc.gatech.edu

Andrea Forte
College of Computing/GVU
Georgia Institute of Technology
801 Atlantic Drive
Atlanta, Georgia
aforte@cc.gatech.edu

Adam Wilson
College of Computing/GVU
Georgia Institute of Technology
801 Atlantic Drive
Atlanta, Georgia
awilson@cc.gatech.edu

ABSTRACT

Introduction to Media Computation is a new CS1 aimed especially at non-majors which was taught in Spring 2003 with some success. The course is contextualized around the theme of manipulating and creating media. Of the 121 students who took the course (2/3 female), only three students dropped (all male), and 89% completed the course with a grade C or better. We attribute the success of the course to the use of a domain context and the process used in designing the course, which involved building upon known issues from the CS education literature and seeking frequent feedback from stakeholders.

Categories and Subject Descriptors

K.4 [**Computers and Education**]: Computer and Information Sciences Education
; H.5.1 [**Information Interfaces and Presentation**]: Multimedia Information Systems

General Terms

Experimentation,Design

Keywords

Multimedia, CS1/2, programming, non-majors

1. A SUCCESSFUL NON-MAJORS CS1 COMPUTING COURSE

At Georgia Institute of Technology (*Georgia Tech*), every incoming student must take a course in computation, including programming. Up until recently, the only class available was our majors-focused CS1 based on the TeachScheme approach[6], which has become one of the most disliked courses on campus among the liberal arts, architecture, and management students. We saw this requirement as an opportunity to build toward Alan Perlis' vision of programming for *all* students—as a component of a general, liberal education [8].

The course that we developed, *Introduction to Media Computation*, is an introduction to computing contextualized around the theme of manipulating and creating media. Students really do program—creating Photoshop-like filters (such as generating negative and greyscale images), reversing and splicing sounds, gathering information (like temperature and news headlines) from Web pages, and creating animations. Details on the design of the course are available elsewhere[1] [11].

The pilot offering of the course in Spring 2003 was successful. 121 students enrolled, 2/3 of whom were women. Only three students dropped the course (all male). 89% of the class earned an A, B, or C in the course. 60% of the respondents on a final survey indicated that they would like to take a second course on the topic. Students wrote eight programs (six collaboratively and two individually) creating or manipulating pictures, sounds, HTML pages, and movies, with some of their programs reaching 100 lines of code.

We do not believe that media computation is the *only* context in which students not traditionally attracted to computer science might succeed. For example, we have described another potential CS1 course organized around information management in which students might build Web harvesting programs and data visualizations [12]. The general approach we are exploring is to use an application domain of information technology that is rich in CS concepts and relevant to the students, then explore introducing computing concepts in terms of that domain. The Media Computation project is a trial and model of that approach.

In this paper, we describe how we designed the Media Computation course, as an example of this process. A sketch of our process follows, and is detailed through the paper.

- *Setting objectives*: We set objectives for the course based on the campus requirements for a computing course, on the national guidelines for CS1, and on the existing computer science education research literature about what students found difficult about computer science.

[1]See also the development website at `http://coweb.cc.gatech.edu/mediaComp-plan`

SIGCSE2004 '04 Norfolk, Virginia, USA

- *Choosing a context*: We select a context that allows us to meet our objectives and helps students to learn programming in a useful context.
- *Set up feedback process*: We sought feedback from faculty in the majors that we planned to serve, as well as from students through on-line and pizza lunch forums.
- *Define infrastructure*: The first stage of defining the content for the course is to choose the language and programming environment, which are critical (and sometimes religious) issues. We found that the process of choosing a language for a non-majors course is as much about culture and politics as it is about pedagogy.
- *Define the course*: Finally, we can define lectures, assignments, and all the details of what makes up a course. Here our decisions were informed by research in the learning sciences [4].

2. DESIGN PROCESS

In the sections below, we detail what we did in each stage of the development process for the Media Computation class. We believe that a similar process could be used to create other contextualized CS1 courses, and perhaps *should* be used, especially when targeting non-CS majors.

2.1 Setting Objectives

The Georgia Tech computing class requirements stated that the course content had to focus on algorithmic thinking and about making choices between different data structures and encodings. Students had to learn to explicitly program algorithms. We also wanted to meet the guidelines set out by *Computing Curricula 2001* [2] in order to create a CS1 that would work at other institutions.

We explicitly chose not to prepare these students to be *software developers*, but instead, to be *tool modifiers*. We do not envision these students as professionals ever sitting down to program at a blank screen. Instead, we imagine them modifying others' programs, and combining existing programs to create new effects. Based on our discussions with faculty in these majors, these students will rarely if ever create a program exceeding 100 lines. The implications of these assumptions and findings are that much of the design content and code documentation procedures that appear in many CS1 curricula are less relevant for these students.

We also explicitly chose to aim the class toward attracting students currently not being retained within computer science, especially women. We used the 2000 AAUW report [1] and *Unlocking the Clubhouse* [14] as our main sources. We set three objectives based on these studies:

- *Relevance*: We decided that we wanted to make the course clearly relevant to this audience. We set an objective to make sure that all the assignments and lectures were relevant to the students' professional goals within that context.

  One implication is that we decided to discuss issues of functional decomposition, how computers work, and even issues of algorithmic complexity and theoretical limits of computation (e.g., Travelling Salesman and Halting problems), but at the *end* of the course. During the first ten weeks of the course, the students are writing programs to manipulate media, and they begin to have questions. "Why are my programs slower than Photoshop?" and "Isn't there a faster/better way to write programs like this?" honestly did arise from the students naturally. At the start of the course, such content is irrelevant, but at the end of the course, it is quite relevant. The implication is that students make many mistakes during those first few weeks that might have been corrected with better functional decomposition, for example. That is a reasonable tradeoff for better learning.

- *Opportunities for Creativity*: One of the comments made by female computer science graduates is that they are surprised to find that computer science offers such opportunity for creativity—it wasn't obvious in the first few courses, but was obvious later [20]. Making more opportunities for creativity in early classes may help improve retention [1].

- *Making the Experience Social*: We wanted students to see computer science as a social activity, not as the asocial lifestyle stereotypically associated with hackers—a stereotype which has negatively influenced retention [14].

2.2 Choosing a Context

Most CS1 curricula aim to teach generalized content and problem-solving skills that can be used in any programming application. Why should we limit ourselves to talking about programming only within a given domain? Doesn't that place a limitation on how our students understand programming?

Research in the learning sciences suggests that, indeed, teaching programming tied to a particular domain can lead to students understanding programming only in terms of that domain. This is the problem of *transfer* [4, 5]. That's why it's important to choose a domain that is relevant to the students. This is not a problem only with students, though—most software experts only can program well within domains that they are familiar with [3]. However, there is strong evidence that without teaching abstract concepts like programming within a concrete domain, students may not learn it at all [13]. Given the track record of CS1 with high failure rates [21], contextualization may offer an important key to improved learning.

One argument that has been made for teaching programming, especially to non-majors, is to teach general problem-solving skills (e.g., [18]). Empirical studies of this claim have shown that we can't reasonably expect an increase in general problem-solving skills after just a single course (about all that we might expect non-majors to take), but transfer of *specific* problem-solving skills can happen [17]. Therefore, teaching programming in a context where students might actually use programming is the best way of teaching students something in a single course that they might use after the course has ended.

The goal of contextualization is to give students relevant, concrete, and understandable examples on which to explain the abstract programming concepts. People learn from examples [13, 4]. While explaining the abstract principles can create an *advanced organizer* for knowledge [4], people tend to reason from concrete and understandable experiences [13].

Within this context, we were able to address our learning objectives. Issues of data structuring and encoding arise naturally in media computation, e.g., sounds are typically arrays of samples, while pictures are matrices of pixel objects, each pixel containing red, green, and blue values. We were able to address the specifics of a CS1 course in the details of the course construction.

Media computation is relevant for these students because, for students not majoring in science or engineering, the computer is used more for communication than calculation. These students will spend their professional lives creating and modifying media. Since all media are becoming digital, and digital media are manipulated with software, programming is a communications skill for these students. To learn to program is to learn how the students' tools work and even (potentially) how to build their own tools. Our interviews with students suggest that they accept this argument, and that makes the class context relevant for them.

We wanted students to have choices in their media for their homework whenever possible to make assignments more creative. For example, one assignment requires the creation of a collage where one image appears multiple times, modified each time. Students get to pick the required image, and can include as many other images as they would like.

The media computation context also provided something to share to encourage a social class setting. We encouraged students to post their media creations in a shared Web space[2], our *CoWeb* tool that we had used in previous computer science courses [10]. We also allowed for collaboration on most assignments, only designating two as "take-home exams" on which no collaboration was allowed. We also used in-class quizzes and exams for assessment, but encouraged collaborative studying including exam review pages and shared answer space on the Web[3].

2.3 Set Up Feedback Process

When we first started planning this class, we created on-line surveys and asked teachers of freshman campus-wide classes (such as introductory English composition, Calculus, and Biology) to invite their students to visit the pages and answer them. Later, as our questions became more specific, we had follow-up surveys just inviting non-CS majors in our introductory computing courses. These gave us important mechanisms for gathering impressions and attitudes, and then for bouncing ideas off of students.

As the class was taking shape, we invited non-CS majors in our introductory computing courses to attend pizza lunches where we presented the class and got feedback on the course. The pizza lunches were not as useful for our development process as the on-line surveys, though. The students tended to be dazzled by the media in the lunch forums, and tended to give us purely positive feedback. On the other hand, the forums helped create an interest in the course, and that spurred more discussion and feedback in the on-line surveys.

We also set up an advisory board of eight faculty from around campus who would review materials and give us advice on what they wanted for their majors. The advisory board was very helpful in several ways. In several cases, the advisory board told us specific content issues that they wanted to see in the course, e.g., one faculty advisor told us about the kinds of graphing that she wanted to see, and another from Architecture suggested a particular topic that was relevant to architects (the difference between vector and bit-mapped representations) that he hoped we could include. The board was also helpful in creating local expertise in the course when it came time for the various academic units to vote whether or not to accept the new course for their majors. It was easy to point at faculty who knew the course better than just what was in the course proposal, and that helped to sell the course to the faculty.

2.4 Define Infrastructure

Our first choice for programming language for the course was Scheme. It's known as a successful first language for many students, and we knew that we could build upon an implementation of Scheme in Squeak [9] that would give us cross-platform multimedia access. Scheme was resoundingly rejected by both students and non-CS faculty. Students saw it as "more of the same"—just like our existing CS1. The faculty rejected it for more surprising reasons: *Because* Scheme is more serious CS. One English faculty member said that she found Scheme unacceptable simply because it was the first language taught at MIT, and that was enough for her to prove that Scheme was unacceptable for her students.

We explored several other languages after that, including Java and Squeak. Java was unacceptable because we used it in our upper-level courses. That branded it as too complex for non-majors. Squeak was simply unknown.

In the end, we settled on Python—in particular, the Jython dialect, implemented in Java, in which we could access cross-platform multimedia easily [19]. Python was acceptable for two reasons:

- First, we could list a number of companies using Python that non-CS faculty recognized, such as Industrial Light & Magic and Google. That was quite important to them, and it does make sense. Non-CS faculty want some measure of quality of materials and content provided for their students, but the non-CS faculty may not have much background in computer science themselves. How can they then vet a programming language for their students? By looking at who else uses it, was the answer we discovered.

- Second, unlike a more obscure language like Squeak, there are references to Python everywhere on the Internet, always associated with terms that the faculty members found consoling: Easy-to-use, Internet-ready, and simple for beginners.

While the choice of language was limited by external factors, we were happy for the choice of Python because of the opportunities it gave us to apply lessons from computer science education research in terms of teaching iteration and conditionals. We know that learning iteration is hard for students [22], but we also know that if that iteration is expressed as a set operation, novices find it easier to understand [15, 16]. Because Python of Python's definition of a `for` loop, we were able to introduce pixel manipulations as a set operation, e.g., `for p in getPixels(picture):`. Later, we introduced a more traditional `for` loop where an index variable varies across a range of integers, but only

[2]See, for example, `http://coweb.cc.gatech.edu/cs1315/440` for postings from the Collage assignment during Spring 2003.

[3]See `http://coweb.cc.gatech.edu/cs1315/194`

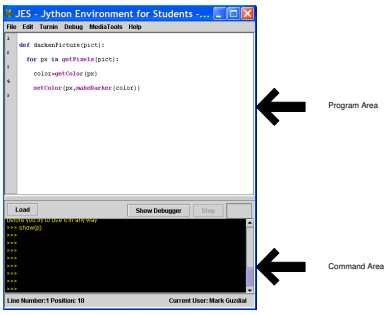

Figure 1: JES programming environment

after students were successfully programming and dealing with iteration at an easier stage.

Once we had chosen our language, we needed to provide tools for this language. We decided to build two sets of tools. The first would be a development environment for the students, JES (Jython Environment for Students) because no such simple IDE existed for Jython. Second, we developed a set of media tools (called MediaTools, implemented in Squeak) to enable students to look at sounds at the sample level, record new sounds, playback movies, and look at individual pixels in pictures. We viewed the MediaTools as important debugging tools for the students.

We modelled JES after the successful Dr. Scheme [6] tools. JES only allowed one open file at a time in the Program Area, and the interpreter was always available for testing and exploration in the Command Area (Figure 1). One of our slogans for the development of JES was "No magic." We wanted JES to be simple to understand. For example, there were no agents to help critique the code, nor visualization tools, nor special debugging modes.

That said, we have found weaknesses in our model. The MediaTools were never used as debugging aids as determined from observations of students while programming. Sitting in a separate application, the MediaTools were simply ignored while students worked on their programs. We have been working to provide some of the media exploration functionality in JES. We have also found it difficult to phrase error messages in a way that makes sense for these students. We are testing a new set of error messages during the Fall 2003 semester with 305 students.

2.5 Building the Course

We then developed the course lectures and assignments to achieve the objectives within the given context and infrastructure. The syllabus[4] for the course walks through each media type, with some repetition of concepts so that conditionals and loops can be re-visited in different contexts.

For example, we introduce pictures as a media type in week 2, including psychophysics issues relevant to media computation (why don't we see 1024x768 dots on the screen?), looping to change colors with the set-oriented `for` loop, conditionals to replace specific colors, then introducing a `for` loop with index numbers to implement mirroring, rotating, cropping, and scaling. Later, in week 4, we introduce sound as a media type, including psychophysics (how human hearing limitations make CD recording possible), looping to manipulate volume (with `for sample in getSamples(sound):`, then indexing by index numbers to do splicing and reversing of sounds.

We explicitly decided not to teach the `else` variation of the `if` statement. Research in computer science education points out that conditionals are difficult to read [7, 22]. By removing the `else` and its implicit test, we require all tests to be explicit and thus more readable. We don't address the decrease in efficiency because that's exactly the kind of issue that is raised as dissuading students in computer science [1, 14].

We do, however, address the encoding issues, such as the number of bits per red, green, and blue channel in the pixel, and the theoretical number of colors that such an encoding provides. We consider this a *relevant* technical detail since representations of color are part of the communications focus of the course, and it allowed us to address the Institute requirements of discussing encoding and data structuring. We similarly discuss the number of bits in a sound sample and sampling rate, and relate that to the limitations of sound recording (e.g., the Nyquist theorem).

Originally, we planned to discuss sound first, before pictures, because the simple one-dimensional array would be easier to manipulate for students than the two-dimensional matrix of pixels. We offered a one day workshop for faculty interested in trying out the course where we used this ordering, and found that the results of the sound examples were too subtle for the students (faculty members) to appreciate. When we moved to pictures, the faculty workshop participants found it much easier to understand. We switched to pictures-first in the class since it allowed for the concrete, visual feedback.

In end of the term surveys, however, some students told us that they preferred sound to pictures, for the simplicity and to match their personal interests. This result confirmed our decision to essentially replicate the progress from set-based `for`, conditionals, and integer-indexed `for` in both pictures and sounds, because it allows students to revisit the issues in different media, to match different student interests.

Student programming assignments built upon the media in relevant communications tasks. The third programming assignment required creation of a collage. A student programming assignment during the week when text manipulation and HTML were introduced was to write a function to generate an HTML index page for all sound and picture files in a given directory. The students' final programming assignment (during week 14) was to write a program to fetch the index page of `http://www.cnn.com`, find the top three headlines from the page, then generate a ticker-tape movie of those headlines.

2.6 Evaluate

We evaluated the course with surveys at three points during the term, interviews with volunteer women in the class, and observations of students using JES to determine usability and strategy errors. We learned that the class was perceived as being relevant by the students, and that students were motivated to want to take more computer science like this class. We also learned of the problems with JES and MediaTools through the observations.

One of the most interesting evaluation lessons was the importance of the social aspect of the course to the students.

[4]The syllabus for the course can be found at `http://coweb.cc.gatech.edu/cs1315/24` and is summarized in [11].

When asked on a final survey what was the most important aspect of the class *not* to change as we went forward with revising the class, nearly 20% mentioned our collaborative CoWeb and over 20% mentioned "collaboration" in general.

3. CONCLUSIONS

The process that we describe in this paper is not specific to designing a Media Computation course for non-CS majors. Rather, we feel that this process is the right kind of process to follow whenever creating a CS course for non-majors. Setting objectives at the start of the process is important. Those objectives helped us determine our context in media computation, though another context may have been perceived with just as much relevancy by the students. Setting up a feedback process is absolutely critical, especially when the target audience is not one's own majors. Infrastructure questions, as we learned, cannot be addressed purely from theoretical or pedagogical reasons—there are important cultural and political questions to address, especially when the students come from other disciplines. The evaluation is important to test one's assumptions and where the weaknesses still are in the design of the course.

The Media Computation course is being continued and grown at Georgia Tech, with over 300 students in Fall 2003 and a planned 450 students in Spring 2004. It is also starting to be adopted elsewhere, with two sections in Fall 2003 at Gainesville College, a two-year institution in Northern Georgia.

4. ACKNOWLEDGMENTS

This research is supported in part by grants from the National Science Foundation (CISE EI program and DUE CCLI program), from the Al West Fund at Georgia Tech, and by the College of Computing and GVU Center. We wish to thank all the students who helped create JES and the Media Computation class, and all the students in the class who volunteered to participate in our studies.

5. REFERENCES

[1] AAUW. *Tech-Savvy: Educating Girls in the New Computer Age*. American Association of University Women Education Foundation, New York, 2000.

[2] ACM/IEEE. Computing curriculum 2001. `http://www.acm.org/sigcse/cc2001`, 2001.

[3] B. Adelson and E. Soloway. The role of domain experience in software design. *IEEE Transactions on Software Engineering*, SE-11(11):1351–1360, 1985.

[4] J. D. Bransford, A. L. Brown, and R. R. Cocking, editors. *How People Learn: Brain, Mind, Experience, and School*. National Academy Press, Washington, D.C., 2000.

[5] J. T. Bruer. *Schools for Thought: A Science of Learning in the Classroom*. MIT Press, Cambridge, MA, 1993.

[6] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. *How to Design Programs: An Introduction to Programming and Computing*. MIT Press, Cambridge, MA, 2001.

[7] T. R. G. Green. Conditional program statements and comprehensibility to professional programmers. *Journal of Occupational Psychology*, 50:93–109, 1977.

[8] M. Greenberger. *Computers and the World of the Future*. Transcribed recordings of lectures held at the Sloan School of Business Administration, April, 1961. MIT Press, Cambridge, MA, 1962.

[9] M. Guzdial. *Squeak: Object-oriented design with Multimedia Applications*. Prentice-Hall, Englewood, NJ, 2001.

[10] M. Guzdial. Use of collaborative multimedia in computer science classes. In *Proceedings of the 2001 Integrating Technology into Computer Science Education Conference*. ACM, Canterbury, UK, 2001.

[11] M. Guzdial. A media computation course for non-majors. In *Proceedings of the Innovation and Technology in Computer Science Education (ITiCSE) 2003 Conference*, pages 104–108, New York, 2003. ACM, ACM.

[12] M. Guzdial and E. Soloway. Computer science is more important than calculus: The challenge of living up to our potential. *Inroads – The SIGCSE Bulletin*, 35(2):5–8, June 2003.

[13] J. Kolodner. *Case Based Reasoning*. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[14] J. Margolis and A. Fisher. *Unlocking the Clubhouse: Women in Computing*. MIT Press, Cambridge, MA, 2002.

[15] L. A. Miller. Programming by non-programmers. *International Journal of Man-Machine Studies*, 6:237–260, 1974. Participants strongly preferred to use set and subset expressions to specify the operations in aggregate.

[16] L. A. Miller. Natural language programming: Styles, strategies, and contrasts. *IBM Systems Journal*, 20(2):184–215, 1981. Languages require iteration where aggregate operations are much easier for novices.

[17] D. B. Palumbo. Programming language/problem-solving research: A review of relevant issues. *Review of Educational Research*, 60(1):65–89, 1990.

[18] S. Papert. *Mindstorms: Children, computers, and powerful ideas*. Basic Books, New York, NY, 1980.

[19] S. Pedroni and N. Rappin. *Jython Essentials*. O'Reilly and Associates, 2002.

[20] S. L. Pfleeger, P. Teller, S. E. Castaneda, M. Wilson, and R. Lindley. Increasing the enrollment of women in computer science. In R. McCauley and J. Gersting, editors, *The Proceedings of the Thirty-second SIGCSE Technical Symposium on Computer Science Education*, pages 386–387. ACM Press, New York, 2001.

[21] H. Roumani. Design guidelines for the lab component of objects-first cs1. In D. Knox, editor, *The Proceedings of the Thirty-third SIGCSE Technical Symposium on Computer Science Education, 2002*, pages 222–226. ACM, New York, 2002. WFD (Withdrawl-Failure-D) rates in CS1 in excess of 30

[22] E. Soloway, J. Bonar, and K. Ehrlich. Cognitive strategies and looping constructs: An empirical study. *Communications of the ACM*, 26(11):853–860, 1983.