SIGCSE 2013 Trip Report - Chris Wilcox

3/9/2013

INTRODUCTION

I am just finishing up with the Special Interest Group on Computer Science Education (SIGCSE), held from 3/7-3/9 in Denver. The conference is being attended by over 1300 faculty, many of whom are either teaching faculty or are doing active research in Computer Science education, or both. I will not give a detailed report on the 18 papers, 2 workshops, 2 birds of a feather sessions, and the keynote speeches, since I doubt anyone is interested in seeing that much information. Instead I will summarize the main ideas circulating in the Computer Science education community, and how they might be applied in our department.

OVERVIEW

Here are the most important ideas that are current in classroom pedagogy. These ideas were repeated throughout the conference. I present the data justifying these techniques and details on how they are practiced below.

INVERTED CLASSROOM: The inverted classroom assigns the reading of textbook and slides in advance of classroom time. Instead of lectures, classroom time is used for interactive activities that engage students, for example, peer instruction or in-class programming activities led by the instructor.

PEER INSTRUCTION: This is the activity that most commonly replaces traditional lectures, i.e. slide presentations. One version of peer instruction involves teams of students using clickers to answer questions designed to stimulate discussion. I attended a workshop by Beth Simon on implementing peer instruction, which I will describe later.

PAIR PROGRAMMING: This is an activity that is the exact opposite of what we practice in our department. Instead of discouraging collaboration, students are paired with other students for programming activities, including lab work and programming assignments done outside the lab or classroom.

MEDIA COMPUTATION: This pertains to the assignment of programming exercises that incorporate graphics, video, and audio in place of having students write console programs that input and output text. There are a number of sources for this material, see MediaComp for more information.

PEDAGOGICAL LANGUAGES: These are programming languages, often paired with graphical environments that are designed purely for academic purposes. Large communities exist that use these in place of traditional languages to simplify the teaching of programming. Examples include Alice, BlueJ, and Scratch.

COMPUTER SCIENCE in HIGH SCHOOL: This is another big topic at the conference. The National Science Foundation and other partners have a goal of putting 10,000 Computer Science teachers into high schools by 2017. There are lots of activities and research related to this goal, including the ongoing effort to define AP curriculum.

UNDERREPRESENTED GROUPS in COMPUTER SCIENCE: Several organizations presented information on trying to improve participation in CS by female and other minority students, including NCWIT, ACM-W, the Anita Borg Institute, etc. In the last decade there has been a decrease in the number of Computer Science graduates from these groups. One frequently cited cause is lack of role models; this is something Elaine is obviously thinking about already when recruiting undergraduate teaching assistants. The final luncheon speaker was Jane Margolis, author of “Stuck in the Shallow End” and “Unlocking the Clubhouse: Women in Computing”, a very interesting speaker.

MOTIVATION

The Computer Science education community has a mound of research related to the efficacy of the various pedagogical practices shown above. Four basic motivations link the disparate techniques:

ENGAGEMENT: The research says that students who are engaged learn more, that is students that are actively participating in constructing solutions are more likely to learn than those passively listening to a lecture. Peer instruction is one of the primary tools used to increase engagement. The videos taken of peer instruction versus traditional lecture format by Beth Simon were very convincing, as was a presentation I saw where they measured the neural activity of students. Neural activity during lectures was comparable to watching television or resting, whereas neural activity during peer instruction and lab activities was very high.

COMMUNITY: The research says that students who belong to a community learn more, especially for groups that traditionally underrepresented in Computer Science. Research shows that students that feel a sense of community in their classes or departments are much less likely to leave Computer Science. Peer instruction and pair programming both create community, and some studies have shown that they increase retention and decrease the number of dropouts. Mentoring is a big part of creating community; many institutions make the assignment of mentors a priority.

COLLABORATION: Research says that collaboration during labs and programming assignments increases the sense of community and improves the outcome. Several speakers claimed to have observed that students that work in groups are much less likely to give up when programming tasks become difficult. This might be something we want to look at, as we currently make every effort to prevent any collaboration in our introductory classes. One paper studied collaborative learning and discovered that in some cases it can produce anxiety and lack of results, but in general they found that the knowledge sharing and synergy that occurs outweighs these disadvantages.

ACCOMPLISHMENT: The keynote speaker, Michael Kölling, said that “experiencing the joy of invention and creation” motivates students. I think this explains why our graphical programming assignments are so popular. He also claims that learning a programming language for the first time entails a high cognitive load and lots of negative feedback! His solution to this is to either avoid the commercial programming languages like C++ and Java in favor of academic languages such as Scratch and Alice, or at the very least to simplify the initial programming experience by providing visual environments such as BlueJ that are much less cluttered than industrial tools like Eclipse. These techniques have apparently been very successful at the K-12 level, however, I would be skeptical that this would be a good idea for our introductory classes.

TEACHING TIPS

I also wanted to report on a number of interesting techniques described in other papers or sessions that I think contained some pragmatic advice for teaching faculty:

1. The *constructive use of errors* was one teaching suggestion. The paper that discussed this topic showed a considerable amount of research supporting the theory that students learn at least as much from incorrect code as they do from correct code. The author of the paper says that errors trigger cognitive conflicts that in turn lead to learning. He presents students with code that contains errors, and makes them fix it, which he claims results in better understanding of difficult concepts like class, object, static, etc. in his introductory Java classes. He also reports that error correction is popular with students.

2. On *how to spark fires when you're not there* was another interesting teaching topic. The researchers studied successful Computer Scientists, both academia and industry and created a list of behaviors that contributed to their success. These included self-learning, exploration and creative play, programming as a means of empowerment, and difficulty stopping leading to an investment of "countless" hours. The interesting part of this study was an experiment where they introduced 40 middle school girls to CodeSpells, a 3D interactive environment. They found that the students exhibited all the same behaviors as successful professionals. I think this is another strong argument for visual programming assignments and other media computation ideas.

3. The *introduction of source control* to students early in the curriculum was cited surprisingly often by different faculty, apparently some teachers use source control in conjunction with automated grading. There was a session sponsored by GitHub in which they talked about the support for academic use of the tool. Free private accounts can be established through github.com/edu, including grouped accounts for courses that allow each student to privately submit and the instructor to view all repositories.

4. I attended Birds of a Feather session on *automatic program grading*, and met Jaime Spacco the creator of Marmoset. Jaime is working on a new tool called CloudCoder that may be of interest to us. Many others in the session are interested in automated tools, but there is not much consensus on what they should do. I did an informal poll of the people there, about 20% use home grown tools, 10% use open source or commercial tools, and the remaining 70% have their teaching assistants do the work manually or with custom scripts for each program. Microsoft claimed to have solved the problem with something called pex4fun, but nobody believed them.

5. The topic of using *test-driven design* in introductory courses still has lots of discussion. I attended a workshop on this where some open-source tools were presented. Marmoset also places lots of emphasis on test-first design.

IMPLEMENTATION

I was especially interested in the details of how peer instruction, pair programming, and media computation were implemented, so I spent lots of time quizzing Beth Simon and I attended her workshop. Here is what I learned:

INVERTED CLASSROOM: This involves giving assignments that should be done by students before class to prepare for the peer instruction sessions. They are accompanied by online tests that force the students to at least scan the material, if not read it in depth. They include reading assignments from the textbook and video made by the instructor. Materials can also include slide presentations to replace lectures, but this is not the prevalent practice.

PEER INSTRUCTION: This departs radically from our teaching format. Only very short presentations are made by the instructor, at most a couple of slides, then 8-10 difficult questions are presented. These are designed to stimulate lots of discussion. A question, usually multiple-choice, is projected and students vote as individuals using iClickers. Then students meet in pre-assigned groups, which persist through the whole term, and they have a discussion in which they try to reach consensus. The question is then repeated and voted on again. Groups are expected to vote unanimously, but this is not enforced. The claim is that this leads to much better results and real learning, mainly because students are required to solve problems, but the group context makes this task easier. After the group vote, the instructor quiets the class, asks for a couple groups to present their answer, then the instructor discusses the result. Occasionally a few slides are shown, but mostly the class goes from one question to the next. Beth's group has a website that provides many examples of good questions. Good questions have very little text, but no obvious answers. They are designed to challenge the students and create discussion.

PAIR PROGRAMMING: Students are randomly assigned partner in the lab, and are given partners with whom they work for the entire term on programming assignments. Something like an enormous Doodle poll is used to match up students so that they have schedule to allow them to meet at least 5 hours per week. Undergraduate tutors grade labs and assignments, mostly without automation, and an interview is part of the process. During the interview individual contributions to the assignment are discussed.

MEDIA COMPUTATION: Beth's group uses assignments from a specific MediaComp textbook developed at Georgia Tech. In addition there are many other resources, and other instructors use video and audio exercises. Many of the projects involve the manipulation of graphics, video, or audio in order to stimulate student creativity. One key is that projects have components such as media that can be personalized by the students. For example, they may incorporate photos of themselves.

RESEARCH OPPORTUNITIES

I identified a couple of opportunities for research papers. The first is that there appears to be a complete lack of standardized curriculums for scientific programing at the undergraduate or graduate level, so a study might be in order. The second is a comparative study on the effectiveness of MyProgrammingLab. I discussed this idea with Beth Simon and she thought it might be promising. I attended a breakfast with Pearson for MyProgrammingLab in which they solicited research in this area.

FUTURE PLANS

Of course what really matters are things that can be incorporated into our introductory course. First, I would like to start trying out some peer instruction in a limited fashion, by converting a handful of Friday classes this semester to see if the student excitement that has been advertised occurs. We already use iClickers to give quizzes on Fridays, so this would not require a huge change. Second, I would like to continue to increase our use of graphical programming assignments, and experiment with a new assignment based on manipulation of audio data. Since we supply support Java classes, this should not require me to increase the difficulty of assignments. Third, I plan on trying to incorporate an exercise in fixing Java code with errors into one or more recitations, to complement our current exercises in interpreting Java programs. Fourth, I think we need to continue along the path of recruiting undergraduate teaching assistants of both genders, both because they seem to handle the teaching duties as well or better than graduate students, and to provide role models for underrepresented populations. Fifth, going forward I would like to revisit our strict policies to deter collaboration, and possibly experiment with peer programming in some form, either in the lab or for programming homework. Some things I am **not** advocating are moving to a non-commercial language like Scratch, nor do I think source control is a good idea for first-year students.

DISCLAIMER

It was easy to get mesmerized at this conference with all of the possibilities for different ways to teach introductory programming courses. I am not sold on all of these techniques, just interested in some of them, and it would clearly be a massive job to completely convert over to all of them. In addition, I think many of the techniques are already in use in upper-division classes, which are by nature of their size more interactive and collaborative, and more prone to having engaged students and strong communities, i.e. research groups. What I am interested in is techniques that make our lower-division classes more effective by helping to decrease failure rates and increase undergraduate retention.