The Five Year Evolution of a Game Programming Course

Gillian Smith and Anne Sullivan
Center for Games and Playable Media
University of California, Santa Cruz
Santa Cruz, CA 95064, USA
{gsmith, anne}@soe.ucsc.edu

ABSTRACT
This paper presents lessons learned from five years of teaching a game design and programming outreach course. This class is taught over the course of a month to high school students participating in the California Summer School for Mathematics and Science (COSMOS) at the University of California, Santa Cruz. Over these five years we have changed everything in the course, from the overall project structure to the programming language used in the class. In this paper we discuss our successes and failures, and offer recommendations to instructors offering similar courses.

Categories and Subject Descriptors
K.3.2 [**Computers and Education**]: Computer and Information Science Education – *Computer science education*

General Terms
Design, Human Factors.

Keywords
Game development, outreach, high school, CS education.

1. INTRODUCTION
Computer games resonate strongly with high school students, with 99% of American teenage boys and 94% of American teenage girls playing video games [7]. Game design is a highly interdisciplinary field, incorporating advanced computer science concepts such as artificial intelligence, computer graphics, and HCI as well as skills from art and design. This paper describes the evolution of a game programming course taught in the California State Summer School for Mathematics and Science (COSMOS) program at the University of California, Santa Cruz. COSMOS is a summer enrichment program for high school students who are hoping to pursue STEM fields when they attend college. In this program, we teach students how to create their own games during an intensive, month-long course. The authors have been involved in teaching the course since its inception in 2007.

COSMOS is split into 9 different clusters based on topic. Each cluster is made up of 18-20 students who are between 9th and 12th grades. Our cluster, *Design of Fun: From Concept to Code*, focuses on teaching computer game development and programming skills. Over the last five years, we have taught 88 students: 40 women and 48 men. These students have a mixture of programming experience, ranging from never having used a computer to having completed AP Computer Science. However, all the students are typically skilled in mathematics and science, and are highly motivated to learn how to create their own games. Some students consider themselves "core" gamers and invest multiple hours per day playing AAA titles (i.e. big budget, high quality games), others play mostly casual online games, and some students express an interest in game design but do not play many games themselves. The goal of the course is to instill in students core CS1 programming skills and a desire to further study computer science and/or game design in university.

Our course focuses equally on teaching fundamental programming skills and providing students with a foundation for how to play, discuss, and analyze games critically. These goals are symbiotic; games have previously been shown as a great motivational force for learning computer science [5][10]. The iterative nature of game design motivates students to reflect on the code they've written, and the procedural thinking and concrete instantiation of abstract ideas that is required when programming feeds back into the iterative game design loop. We have previously described the structure of the game design portion of the class and our recommendations for structuring a game design course [15]; this paper focuses entirely on the programming portion of the course, called *Technologies of Fun*.

The focus on creating a project in a short time period, as well as the variety in experiences and backgrounds of our students provides a unique set of challenges and opportunities for us as instructors. We have tried a number of different teaching methods, learning a great deal about how to effectively teach game programming to a high school audience. We will look at these throughout the paper, presenting what we have learned over the five years this program has been offered. We will first explain changes made in the design of the course itself: the programming languages used, programming instruction methods, and some of the techniques we have used to effectively teach to a mixed-skill classroom. We will then discuss the project portion of the course: dealing with student expectations, assignment format, and prototyping. Finally, we present our recommendations for other instructors involved in similar efforts to teach both game design and programming.

2. COURSE DESCRIPTION
The programming portion of the game design cluster, *Technologies of Fun* is an interactive, projects-based course in which the students program their own small games. The course is designed to teach students the skills required to design and develop their own game over the course of the month-long class. We assume that students entering the course will have no prior programming knowledge; therefore, the goal is to give them a fast-paced course in fundamental programming skills needed for game programming. *Technologies* also includes some short lectures that discuss more advanced concepts, such as 3D graphics and artificial intelligence, most of which will not be used in the students' games. The aim of this is to get students excited about computer science and interested in learning more when they get to college.

SIGCSE'12, February 29–March 3, 2012, Raleigh, North Carolina,USA.

2.1 Class Structure

Technologies of Fun is taught in a three-hour block, four days per week for the four weeks of the program. The remaining time in the week is taken up with the game design course [15], and a transferable skills course that teaches research and presentation skills. We took an activities-based approach when teaching programming. Students had access to laptops during lectures, and the first hour of lecture was structured to introduce a new programming concept with brief breaks during which students could experiment with the new material. The other two hours were spent with students working on the projects while the instructor and teaching assistants (TAs) supervise and answer any questions.

Students are encouraged to share their progress with the class by presenting their work at various milestones, or showing how they have solved a particularly challenging problem. They could also share code with each other through a cluster wiki[1], which served as a home for lecture materials and assignment descriptions.

In the last two weeks of the program, there is very little time devoted to lecturing, as students need to work full time on their projects. Lectures become more impromptu: they are only held when it becomes clear that a number of students are struggling with the same general problem, such as adding music to their game, or implementing side-scrolling animation.

2.2 Project Structure

As a requirement for completing the COSMOS program, students must present a project to their teachers and classmates at their home institution. To accommodate this, the majority of our program is focused on enabling the students to produce a complete game. The games are created entirely by the students using programming and design concepts that they have learned in class. All of the games that students create are 2D games with original graphics. Examples of student-produced games include a puzzle platformer where the player shifts between two different worlds; a space-themed, column-based, tower defense game; a game where the player uses the Wiimote to control a net for catching ghosts in a graveyard; and a puzzle game where the player must shape the path of sushi around various obstacles from a refrigerator to a hungry boy.

The project is structured as a series of deliverable assignments. The first assignment is a game pitch in which the student describes a general setup of the game, a narrative hook and a brief overview of the game interaction. Game pitches are used to assign teams, and the teams decide which of their game pitch or combination of pitches they will pursue. The next assignment is for the team to create a design document which discusses their proposed project in much more detail. The students were then required to take the core mechanic (main type of interaction with the game) and create a paper prototype to test this mechanic. A paper prototype is a paper-based representation of the game, and is created to quickly test concepts. Once the core mechanic has been tested and adapted through the paper prototype, the students are then allowed to create a computational prototype. From there the students expand and create their fully realized game.

3. COURSE EVOLUTION

In the last five years we have taught three different programming languages and game engines and changed our teaching methodology. Each year we have reduced the amount of time we are lecturing, and increased the amount of time the students participate in activities, giving even more activity time closer to the end of the course when they focus on their projects. Lectures are also more interactive, with the instructor programming live in front of the students, so they can follow the process and see how small code changes lead to different results.

3.1 Programming Language Choice

We have used three different programming languages in the five years of the program: Python with the Pygame library[2] in 2007 and 2008, Java with the Greenfoot [6] framework in 2009, and finally Processing [12] in 2010 and 2011. We intend to continue using Processing in this course for the foreseeable future. There were many different factors that played into our decision for which programming language to use, including ease of learning, applicability to future computer science courses, and engine support. We also had to bear in mind that most students in the class would have never programmed before.

Visual programming environments such as Scratch [13], Alice [2], and GameMaker[3] are used by many programs that are similar to ours [3][1][18]. There are a few different reasons that we have chosen to use a traditional language instead. Firstly, one of the goals of COSMOS is to give students an experience similar to what they will have in a university, where traditional programming languages are usually used in introductory courses. We also felt that many of the well-established visual environments are targeted at audiences younger than high school students; Scratch and Alice in particular are have audiences as young as elementary school students [17]. Finally, we wanted students to learn programming skills that could be easily applied to domains outside of game design or interactive graphics.

3.1.1 Python and Pygame

Python was initially chosen due to Pygame's support for simple 2D games and recent reports of Python's suitability for introductory programming [11]. Its interactive shell is particularly useful for teaching simple programming concepts such as mathematical operators, variables, and lists. The original instructor for the course also built a game engine that sat atop Pygame, which added support for sprites, animated sprites, collision detection, Bluetooth-connected gamepads and Wiimotes, and particle systems. For the two years that we used this setup, we found that students with no programming experience could make their own animated scenes in the first week of class. However, by the beginning of week three, most students were struggling with their projects. Many of them started their games writing code without a complete understanding of how it worked; as their understanding caught up with them, they would feel overwhelmed by the poor design decisions they had made earlier in the project.

We can expect this to happen for introductory students using any programming language; however, Python had some disadvantages that exacerbated the problem. While it's a fairly simple language to learn, Python is overly supportive of the programmer making mistakes. For example, we frequently found students who had defined functions inside of while loops and then received confusing scoping errors, or who didn't understand that identifiers are case sensitive, and so they were assigning values into different variables than they were reading later. It became clear that we

[1] http://www.sokath.com/cosmos2011/doku.php

[2] http://www.pygame.org/

[3] http://www.yoyogames.com/gamemaker

needed to use a language that provides more structure and the safety of static typing and explicit variable declaration: the compiler can provide much more rapid feedback than periodic support from TAs.

3.1.2 Java and Greenfoot

In 2009, in response to the problems we faced with Python, we decided to use Java as our programming language, supported by the Greenfoot framework for making games. Those students who came into class with programming experience had typically learned Java in their high school computer science classes, and we expected that any students who continued studying computer science after the program would be most likely to learn Java in their introductory courses. Greenfoot offered a number of helpful features for students: ease of drawing graphics, mouse and keyboard callbacks, and collision detection are all built-in. There was an observed improvement in the quality of code the students were writing in Java when compared to Python, as the compiler could catch more errors and give helpful messages pointing to where the error occurred.

However, there were a number of problems introduced by the use of Greenfoot as well. Many students were confused by heavy Java syntax such as import statements, class declarations, and especially the type-casting required for collision detection. Greenfoot also forced us to teach object-oriented programming and the concept of inheritance from the first day of instruction. This proved too abstract and difficult a concept, especially for those students who had limited experience with using a computer.

A benefit of using Python that we lost moving to Greenfoot was that students could write simple procedural code for drawing an image on the screen with only a few lines of code, and could immediately see the results of their work. To achieve the same effect in Greenfoot required editing two different files: the *world* that the object would live in, and the *actor* itself. Students also felt confused over the graphical interface accompanying the scenario; it proved helpful for testing how objects interacted with each other, but students were confused over when they could use that interface versus when they needed to programmatically specify the location of different objects.

3.1.3 Processing

These lessons led to our decision to use Processing starting in 2010. Processing is a Java-based language and environment that is intended for creating interactive art and rapid prototyping. It was clear from 2009 that a Java-like language is an appropriate choice for the course, but we needed a simpler point of entry. Processing makes it easy to draw shapes and pictures to the screen and have a user interact with them: it takes a single line of code to draw a rectangle to the screen, and two lines to make that rectangle be a different color. The availability of many different small art projects written in Processing[4] also encouraged students to experiment; for the first time in teaching the course, we allowed students to take code from other sources (with appropriate attribution) and use it in their games. We only required that they be able to loosely explain how it worked. This kept students excited during class, which motivated them to learn not only from their instructors but also from each other. We saw many instances of one student learning how to make a certain effect in their program and then that effect appearing in different variations across other students' programs.

[4] http://www.openprocessing.org

Processing also has the benefit of allowing a gentle introduction to object-oriented programming, just as Python provided. Students who were not prepared to understand more complicated programming concepts could opt-out and use purely procedural programming; those who were ready to use objects (often those who had already taken a computer science course) could do so with no problems.

3.2 Programming Instruction

This section discusses the various approaches we've used, from unstructured programming activities to highly structured mandatory assignments. Our main goal has been to balance the students' desires to be creative with a structure that ensures everyone learns the core programming concepts.

3.2.1 Freeform Programming

In 2007 and 2008, we took a freeform approach to the programming course. We did not give any assignments; rather, the first week was spent teaching core programming concepts which students could apply however they wished, followed by three weeks of individual and small group instruction as needed, while students worked on their games. In the first week, many students chose to make small animations as introductory scenes for their final games; others began haphazardly constructing pieces of a game, such as movement logic and simple collisions. All of the students had code that they had written on the first day of class in their final games.

This approach to teaching introductory programming has a number of disadvantages, many of which seem obvious in retrospect. Perhaps the most problematic result was that students were stuck with any mistakes or poor design decisions made early on in the project. Students were reluctant to throw away their work and start afresh, even when encouraged to do so by instructors, due to concern that they would not be able to re-create their efforts.

However, there was also one major advantage to this lack of structure: students were continually motivated to learn, due in large part to the creative control they had over their projects [14]. For example, instead of learning about control flow in the abstract, they learned about it because they needed to solve a particular problem they were having in their project.

3.2.2 Mandatory Mini-Game Assignment

In 2009, we decided that the disadvantages of a freeform project far outweighed the advantages, and added a programming activity at the beginning of the class, which students were required to complete before they could start working on their game projects.

This small project was the same for each student: a simple game where the player controls a hippopotamus moving around a maze collecting all the flowers before leaving. The game taught students almost all the major concepts they would need to be able to build their own games: drawing, keyboard input, collision detection with maze walls and flowers, variables to store information about player state, functions and classes. This was fairly successful: by the time students were making their own games, they could look back on their old code to see patterns for how to solve their current problems.

However, the students were not nearly as excited or motivated by the example game as they had been in previous years. Creativity and ownership over their work was clearly very important to them.

3.2.3 Creativity in Small Assignments

In 2010 and 2011 we combined our two prior strategies by instituting a required individual project in the first week, but allowing students to choose what that project was within certain constraints: it had to involve keyboard or mouse input, at least three objects had to exist on screen and those objects must be connected in some way, at least one of the objects was required to move around the screen, and the objects must change color over time. Students generally enjoyed working on their "three points in space" assignment, but still learned all of the major techniques they would need for making their games.

We also introduced a second assignment to combat a common problem we had seen in prior classes—students did not comment their code, and struggled to integrate different sections of code with each other. The second assignment was to modify somebody else's three points assignment in a meaningful way, e.g. adding a new object that interacted with existing ones, changing the colors, or changing how objects moved around the world. The students were not allowed to communicate with each other for this assignment; they were required to declare the change they wanted to the instructors, then read their classmate's code and make the modification themselves. This assignment was extremely effective in teaching the value of commenting code, and seemed to improve each student's understanding of programming concepts by forcing them to read and understand code not in their own style.

Only after completing these two individual assignments were students allowed to work together on their games. While this meant that students started work on their game projects half a week later than in prior iterations of the course, we feel that their improved programming skills made up for the loss of time.

3.3 Handling a mixed-skill classroom

COSMOS accepts high school students from all over California. Our youngest students have been only 13 years old, and our oldest have been 18 years old. This diversity in students naturally leads to a mixed-skill classroom in terms of both programming and mathematics ability, although we do require that all students have completed high school Algebra II. Some students, particularly from the less affluent areas of the state, have no access to computer science courses and limited access to computers. Others have already taken an Advanced Placement (AP) Computer Science course and are looking for further challenges.

Over the five years of our program, we have used a number of different techniques to teach to a mixed-skill classroom. A successful approach has been to create additional content for advanced students to work through while others are learning the basics of programming. The students are encouraged to quietly work through the activities on these handouts while the rest of the class is listening to the lecture, and ask their questions during the group's programming time. Activities have included drawing with splines, the basics of 3D graphics, and programming an artificial intelligence to play Tic-Tac-Toe against a human.

In 2011, we introduced an activity called "Make It Fun", based on a similar activity suggested by Steve Swink [16]. We provided students with source code for a simple maze game with procedurally generated levels. The player controls a ship and navigates the ship around the maze with the arrow keys, collecting coins. This activity was designed to appeal to students at three different skill levels. Those who were just getting started learning programming could read the source code, but choose only to change the values for variables controlling the appearance of the maze, the number of walls and coins in the maze, the size of the ship, and how quickly the ship moves. Intermediate students could choose to modify behavioral aspects of the code, such as changing the control scheme for the ship or adding code for what happens when the player wins or loses. More advanced students could delve into the procedural level generator, which required an understanding of three-dimensional arrays, loops, and random generation. The students greatly enjoyed this assignment as they were in control of making their first game, and the ability to immediately see changes to a game based on even the smallest code changes was a useful learning exercise.

4. PROJECT EVOLUTION

We have also made numerous changes to the design of the class project, to meet the goal of having students spend more time in the design phase before committing their ideas to code, and to ensure that every student learns how to program. To accomplish this, we added multiple prototyping steps to the project and instituted rules about when students were allowed to focus on the aesthetic aspects of the project such as art and music.

4.1 Calibrating Student Expectations

Each year we have become more emphatic about setting up the expectations of the students. Many of the students expect to create AAA-like games using some of the industry-standard 3D game engines. Given time constraints and the amount of learning required, it is more likely that they will instead create a far simpler, unpolished 2D game. Unless expectations are calibrated early, there is a certain amount of frustration and growing disappointment that happens as the student progresses through the course.

In 2007, our first year, we mentioned our expectations as we assigned the game project in the second week of the course. In 2011, we told students at the first day orientation and showed examples of previous games on the first day of class to allow them time to realign their expectations before they were required to create a design document. We found that the game designs showed more creativity and innovation in game mechanics and puzzles instead of being focused on realistic graphics or physics. Keeping the students focused on a 2D game also discouraged re-implementation of their favorite games, which was a common issue in the first and second years of the program.

4.2 Physical Prototypes

One of the major changes we made to the class is a move from the traditional waterfall method to an iterative design process [19]. A key concept in the iterative design process is the use of prototypes to be able to playtest the game design at many stages during game creation. We first introduced a computational prototype (discussed in Section 3.4), which was quite successful. Building on this success, in 2010 we added a requirement to build a playable physical prototype to the project.

When a prototype was not required, students would begin coding directly from their design documents. This would often lead to a situation where the students would be unable to make changes to their code in the last week as the code had evolved in a way that made it quite brittle. Building a physical prototype requires the students to think through mechanics of the game instead of designing on the fly while they were programming. By requiring a physical prototype even before coding, it meant that the students designed the code in a much more stable fashion as they reflected on concrete play experiences. The design would still change as they coded, but they were approaching the code in a much more structured way.

Physical prototypes have a couple of benefits. Firstly, they are simple to build which allows rapid iteration and refinement. When the students did not build prototypes, they would work through a design idea from start to finish without being able to try out their game mechanics until the game was mostly complete. This left very little time and flexibility for changes if the mechanics did not work out in practice. With the prototypes, students were able to try out and modify their game mechanics multiple times before they ever started coding.

Physical prototyping also allowed the students to make design progress while still learning the programming language. Each prototype was played by other students in the class, who gave feedback on parts that they found confusing or difficult. Most students found that they needed to make changes to their game design to address the issues the other students had, and all students made changes purely based on their experience designing the prototype.

4.3 Computational Prototype

Our students often have difficulty with understanding where to start with making their games. In the first two years of COSMOS we addressed the problem by asking students to list all of the tasks required for them to complete their games, and then assign a priority to each of these tasks. Instructional staff would then go through these lists with the students, adding further details and commenting on the stated priorities. However, even with instructor assistance, the list was always incomplete, unwieldy, and rarely referred to after its creation.

In 2009, we adopted Fullerton et al.'s recommendation of digitally prototyping early in the design process [4] by instituting a computational prototype assignment. We required students to identify a core mechanic of their game and prototype it. They were required to complete this prototype and show it to instructors, and then create a list of remaining tasks on the game. We found that by requiring students to make a concrete implementation of their core game design, teams were able to better communicate their ideas to each other and all team members were on the same page throughout the rest of the project. This reduced group tension due to creative differences.

There were also a number of cases where building the prototype changed the direction of the game. For example, one game made in 2010 began in a written concept as a rogue-like dungeon crawler where the character had a limited field of view. After computationally prototyping their movement mechanics, the group decided to change their game to better explore this single mechanic by turning it into a game of exploration and collection rather than adding a number of other mechanics to create a turn-based dungeon crawler with fog of war. Although the redesigned game was much simpler, it allowed the students the opportunity to fully explore this mechanic through many different level designs, rather than shallow exploration of many different mechanics.

4.4 Ensuring equal division of work

One of the goals of our cluster is to have students create a game that they can be proud of and feel that they own; to this end, we found it important to ensure that all of the art and music in their games were made by themselves or their peers. We cautioned against using freely available art from internet sources and advised that even if they felt their art was poor, their games would look better with consistent amateur art versus inconsistent professional art. In the first three years, this had the unfortunate side effect that many teams began worrying about art assets for their games entirely too early in the design process–frequently before core game concepts had even been determined. It also led to a seemingly natural split in teams, where one member would become responsible for art and another responsible for code. Additionally, while mixed-gender teams constituted only about 20% of the overall teams every year, in each mixed-gender team the female student was always the artist and the male student was always the programmer. This situation was extremely concerning; in a computer science focused course, half of our students weren't practicing any programming, and women, a highly under-represented group in computer science [9], were being left out of programming activities.

In 2010 we instituted a requirement that no work be done on the art or music before the final five days of class. While this decision met with resistance at first, students quickly embraced the notion of using programmer art -- rectangles and circles -- and instead exploring their game designs more deeply. The "no art" requirement, combined with assigning teams such that students in teams had equivalent skill levels, was extremely effective in addressing this problem. With no clear division of labor, the inexperienced students were naturally inclined to participate in pair programming, which has been shown to improve results in introductory students [8], and more advanced students learned how to efficiently divide programming tasks. It also meant that each student had continuous input over the evolution of their game, improving their sense of ownership over the final product. In 2011, we relaxed this requirement and allowed students to create art and music earlier in the course, but only during their homework time. This allowed students to make incremental progress on the art for their game and experiment with different animation techniques while also requiring all students to be involved in programming.

5. DISCUSSION

Based on our experiences developing this course, we have drawn up a set of concrete recommendations for similar game design programs to follow:

A programming language should be chosen that minimizes programmer errors and maximizes creative potential. We found the most success when students used a language that can catch the majority of their programming errors at compile-time. The language should also support ease of creative experimentation. For example, a clear benefit we saw in Processing was how easy it was to get started with drawing shapes to the screen and having those shapes move; this framework meant that students could learn how to program through experimentation, such as changing the color and movement patterns of shapes. Later, this led to an ability to focus on testing out different game mechanics, thus maximizing their creative potential.

An iterative approach to game design allows students to build better games more efficiently. Although prototyping is a time-consuming process, our experience has shown that it is worth requiring prototyping stages early in the students' design process; prototyping not only helps to refine the game design, it also ensures that students are communicating their ideas effectively and that everyone in the group understands what they are making.

Design activities and assignments to have multiple entry points to support learning in students at multiple skill-levels. The "make it fun" activity described in Section 2.3 was hugely successful in teaching programming concepts to all students because it had multiple entry points, from editing variable values

to changing complex behavior. Similarly, the three-points assignment is successful because it allows every student the opportunity to work at their own pace; beginning students can complete simple versions of the assignments while more advanced students can explore their creativity further. Giving students creative control over their assignments motivates them to learn more. For example, instead of being taught about if statements in the abstract, students typically ask the instructor how they can make a shape turn around when it hits the edge of the screen.

5.1 Future Plans

Technologies of Fun suffers from a strict time limit: four weeks of classes, even at 3 hours/day, is an extremely short period in which to teach programming skills and mentor students creating their games. This time limitation stops us from making some desired changes to the course, such as requiring students to make multiple computational prototypes that stand separate from their main game code. However, we would like to experiment with requiring two separate, smaller game projects as opposed to a single large project. The majority of the code for the game is written in the middle two weeks of the course—the beginning is devoted largely to teaching game programming, and the end is devoted to polishing games and presenting them to their peers. Also, much of the programming effort in the latter half of those two weeks is in fixing bugs introduced by naïve design choices made in the first few days. By creating two games, students would have the opportunity to learn from their mistakes in the first game and apply those lessons to a second game.

This paper has presented our lessons learned from five years of iteration designing and teaching a month-long intensive game programming course. As the program has matured, we believe it has become much stronger, with more innovative games produced by more prepared students. We hope that the lessons presented here will help other instructors in designing their own outreach programs and courses that use game design to teach computer science concepts.

6. ACKNOWLEDGMENTS

COSMOS is a program under the UC Office of the President, run by the Educational Partnership Center, and funded by both the state of California and by private sponsors. The authors would like to acknowledge the efforts of Jim Whitehead (UCSC), Nathan Whitehead (NVIDIA), and Heather Logas (UCSC) as former instructors for the Structure of Fun and Technologies of Fun courses, respectively. Finally, we would like to thank the cluster's Teacher Fellows, Takeshi Kaneko and Thomas May, for their insight into teaching methodology, and all of the UCSC COSMOS staff for their support.

7. REFERENCES

[1] Carmichael, Gail. Girls, Computer Science, and Games. *SIGCSE Bulletin*, 40, 4 (2008), 107-110.

[2] Cooper, Stephen, Dann, Wanda, and Pausch, Randy. Teaching Object-First in Introductory Computer Science. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education* (Reno, NV, USA 2003), 191-195.

[3] Doran, Katelyn, Boyce, Acey, and Finkelstein, Samantha. Reaching Out with Game Design. In *Proceedings of the Fifth International Conference on the Foundations of Digital Games (FDG '10)* (Monterey, CA, USA 2010), 250-251.

[4] Fullerton, Tracy. *Game Design Workshop: A Playcentric Approach to Creating Innovative Games*. Morgan Kaufmann, Burlington, MA, USA, 2008.

[5] Guzdial, Mark and Soloway, Elliot. Teaching the Nintendo Generation to Program. *Communications of the ACM*, 45, 4 (April 2002), 17-21.

[6] Henriksen, Poul and Kölling, Michael. greenfoot: Combining Object Visualisation with Interaction. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (Vancouver, BC, Canada 2004), 73-82.

[7] Lenhart, Amanda, Kahne, Joseph, Middaugh, Ellen, Macgill, Alexandra, Evans, Chris, and Vitak, Jessica. *Teens, Video Games and Civics*. Pew Internet, Pew Research Center, 2008.

[8] McDowell, Charlie, Werner, Linda, Bullock, Heather, and Fernald, Julian. The Effects of Pair-Programming on Performance in an Introductory Programming Course. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education* (Cincinnati, KY, USA 2002), 38-42.

[9] NCWIT. *The NCWIT Scorecard: A Report on the Status of Women in Information Technology*. URL: http://www.ncwit.org/scorecardflash/, 2010.

[10] Overmars, Mark H. Teaching Computer Science through Game Design. *IEEE Computer*, 37, 4 (April 2004), 81-83.

[11] Ranum, David, Miller, Bradley, Zelle, John, and Guzdial, Mark. Successful Approaches to Teaching Introductory Computer Science Courses with Python (Special Session). In *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education* (Houston, TX, USA 2006).

[12] Reas, Casey and Fry, Benjamin. Processing: A Learning Environment for Creating Interactive Web Graphics. *ACM SIGGRAPH 2003 Web Graphics* (2003).

[13] Resnick, Mitchel, Maloney, John, Monroy-Hernández, Andrés et al. Scratch: Programming for All. *Communications of the ACM*, 52, 11 (November 2009), 60-67.

[14] Romeike, Ralf. Applying Creativity in CS High School Education - Criteria, Teaching Example, and Evaluation. In *7th Baltic Sea Conference on Computing Education Research* (Koli National Park, Finland 2007).

[15] Sullivan, Anne and Smith, Gillian. Lessons in Teaching Game Design. In *Proceedings of the 2011 International Conference on the Foundations of Digital Games (FDG 2011)* (Bordeaux, France June 2011).

[16] Swink, Steve. *Tune: A Game about Game Design*. September 2006. Retrieved September 2, 2011 from http://www.steveswink.com/tune/

[17] Utting, Ian, Cooper, Stephen, Kölling, Michael, Maloney, John, and Resnick, Mitchel. Alice, Greenfoot, and Scratch -- A Discussion. *ACM Transactions on Computing Education*, 10, 4 (2010).

[18] Werner, Linda, Denner, Jill, Bliesner, Michelle, and Rex, Pat. Can Middle-Schoolers Use Storytelling Alice to Make Games? Results of a Pilot Study. (Orlando, FL 2009), Proceedings of the Fourth International Conference on the Foundations of Digital Games (FDG '09), 207-214.

[19] Zimmerman, Eric. Play as Research: The Iterative Design Process. In Laurel, Brenda, ed., *Design Research: Methods and Perspectives.* MIT Press, Cambridge, MA, USA, 2003.