Lessons from COSMOS: Four Years of Iteration on a Game Design Class

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

ABSTRACT
In this paper, we describe a video game design and programming class offered as part of the California Summer School for Mathematics and Science (COSMOS) program at University of California, Santa Cruz. The authors have co-taught this class for four years, and have had an opportunity to iteratively explore a number of approaches in teaching the class. The course has evolved a great deal over the last four years; this paper presents a post-mortem of the program so far, describing our different approaches, our successes and failures, insight into the game design process for students, and recommendations for future game design courses.

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

General Terms
Design, Human Factors, Theory.

Keywords
Games, outreach, high school, computer science fluency, lesson plans, education, project-based course, game design and development.

1. INTRODUCTION
Each summer, the University of California at Santa Cruz (UCSC)—along with three other University of California campuses—hosts the California State Summer School for Mathematics and Science (COSMOS) program. COSMOS is a month-long hands-on program for high-school students, focused on exposing students to university-level topics and projects in the Science, Technology, Engineering, and Mathematics (STEM) fields. Each campus provides eight or nine *clusters* within the program, each concentrating on a different STEM field. For the last four years, UCSC has offered a cluster called "Video Games: The Design of Fun, from Concept to Code," which gives the students background on designing and programming games, as well as giving them the opportunity to create their own video game. We have been involved with the cluster since its inception, as teaching assistants for the first year, and then as instructors for the later years.

The video game cluster is made up of 18-20 students who have just completed somewhere between 8th and 12th grades. The class is made up of approximately equal numbers of boys and girls, ranging in age from 13-18. When students apply to the COSMOS program, they provide their top three choices for clusters. The students who choose our cluster therefore have a stated interest in game design, although their backgrounds can be very different. While some students may have taken a computer science course in high school, most students who enter the program have no prior programming experiences, and there are some students who have rarely used a computer at all. Some students consider themselves core gamers, while others have an interest in game design but do not play many games themselves.

As part of a STEM enrichment program, our cluster is heavily computer-science focused; one of our primary goals is to ensure that each student learns fundamental programming skills. However, we are equally focused on providing our students with a foundation in how to play, discuss, and analyze games critically, and teaching them about the game design process. We feel that these goals are symbiotic; games have previously been shown as a great motivational force for learning computer science [10][18]. 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.

The students attend two daily classes which are taught by different instructors. The first, *Structure of Fun*, teaches the theory and techniques for game design, while *Technologies of Fun* focuses on teaching the programming and computer science concepts. For the sake of clarity, these classes will respectively be referred to as *Structure* and *Technologies* for the remainder of the paper.

The *Structure* class provides an interdisciplinary overview of the design principles used to create fun and engaging video games. Students participate in discussions about theories of fun and play, as well as learn how to apply these theories to their own projects. The course also explores games that address social and humanitarian issues, and games that are designed to appeal to different audiences. Students explore these issues by examining existing games and by working alone and in teams to create two playable paper-based game designs.

As the programming portion of the game design cluster, *Technologies* is an interactive, projects-based course in which the students program their own small games. We assume that students entering the course will have no prior programming knowledge; therefore, the goal is to give them a crash course in fundamental programming skills. *Technologies* also includes a few short side-lectures that talk about more advanced concepts, most of which will not be used in the students' games, with the aim of getting students excited about computer science and interested in learning more when they get to college. Advanced topics include 3D graphics and artificial intelligence techniques.

As a requirement for completing the COSMOS program, students must present their final game projects to their teachers and classmates at their home institution. The majority of our program is therefore focused on the production of a complete game, and we have experimented with a number of different teaching strategies for ensuring that students make the best game they possibly can while also teaching a foundation in game analysis and core programming concepts that can be applied outside of game

creation. 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 focus on creating a project in a short time period, as well as the variety of 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 design and computer science to a high school audience. We will look at these throughout the paper, presenting what we have learned over the four years of offering the program. We will first look at the design of the course itself: the programming languages used, as well as the design methodology, and some of the techniques we have used to deal with a mixed-skill classroom. We will then discuss the project portion of the course: dealing with student expectations, creating a theme for the students to work around, team selection, assignment format, and prototyping. Finally, we present what we plan to change in future years of the COSMOS program, and our recommendations for other instructors involved in similar efforts to teach both game design and programming.

2. COURSE DESIGN

We are fortunate that we have a large amount of latitude with the classes we teach. This allowed us ample opportunities to experiment with different design methodologies and programming languages. Over these first four years of our program we have taught three different programming languages and two design methodologies. The classes also evolved from being mainly lecture-based to incorporating significantly more activities for the students.

2.1 Class Format

The classes are broken up into two three-hour blocks, with a break during each class, and a lunch break between classes. While there is easily enough material to fill a month worth of three-hour long classes, we found that some students struggled with successfully incorporating lecture material into their projects from lecture alone. Therefore, we began to incorporate activities to apply the knowledge they were learning in the lectures to help connect these abstract concepts to their games and projects. After integrating more activities, students began independently incorporating the design concepts into discussions about their projects as well as other games.

To integrate the activities, each *Structure* class was started with a game demonstration. The game would be selected to highlight the topic of the class for that day (such as *One Button Bob* [2] to discuss game mechanics or *Flower* [24] to illustrate a focus on aesthetics), and was typically chosen from less well-known video games or even board games. This gave the students chances to see games they wouldn't normally come across, and in some cases, challenged their definition of a game. For example, some students had only encountered conflict-heavy games, and an exploration-based, sensory game like *Flower* sat outside of their preconceived notions of what a game is. During the lecture, the games were referenced and tied into the material, and at the end of the lecture, the students were given a design task which incorporated the concepts they had learned.

Figure 1. A board game created by the students, designed around the aesthetic of discovery. Players discover new planets based on the cards they draw.

Technologies has a similar activities-based approach when teaching programming. Each student in the class had access to the same laptop for the duration of the course. The first hour of the three hour class was devoted to a lecture that introduces a new programming concept, with brief breaks during which students can 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, which served as a home for lecture materials and assignment descriptions.

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. For example, *Structure of Fun* replaced long lectures about particular game design theories with activities where students explore the concept, such as designing a small board game for a specific aesthetic (see Figure 1). *Technologies* 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.

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 problem, such as adding music to their game, or implementing side-scrolling animation.

2.2 Programming Language Choice

We have used three different programming languages in the four years of the program: Python with the Pygame library[1] in 2007 and 2008, Java with the Greenfoot [11] framework in 2009, and finally Processing [20] in 2010. 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

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

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 [21], Alice [5], and GameMaker[2] are used by many programs that are similar to ours [8][4][26]. 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 [25]. Finally, we wanted students to learn programming skills that could be easily applied to domains outside of game design.

2.2.1 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 [19]. 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.

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 variable than they were reading later. It became clear that we 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.

2.2.2 Java and Greenfoot

In 2009, 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.

2.2.3 Processing

These lessons led to our decision to use Processing 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[3] also encouraged students to experiment; for the first time in the four years of the program, 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 students 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.

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.

2.3 Handling a mixed-skill classroom

COSMOS accepts students from all over California, and allows anyone who is entering the 9th grade through leaving the 12th grade to apply. 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

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

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

Placement (AP) Computer Science course and are looking for further challenges.

Over the four years of our program, we have used a number of different techniques to teach to a mixed-skill classroom. By far the most 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.

The challenges of teaching to a mixed-skill classroom were also a large motivator for moving to instructors assigning teams rather than students self-selecting teams, as discussed in Section 3.3. We attempt to make sure that students in the same group are at an approximately equivalent skill level; without this, our experience indicates that even the best-intentioned teams end up having the best programmer do all the work, with the less advanced student only watching without engaging actively.

2.4 Design Formalism Choice

We have taught two schools of game design over the four years of the program. Initially, we focused on theoretical design concepts such as Salen & Zimmerman's theories of the magic circle and game rules [23], Juul's definition of a game [14], and Crawford's definition of interactivity [7]. In 2010, we introduced the Mechanics, Dynamics, Aesthetics (MDA) framework [12] as a practical framework to incorporate with the previous theories.

2.4.1 Theoretical Design Concepts

When we began the program, much of the game design material was borrowed from our college-level game design course. The material focused on a definition-based approach to game design. This method gives the students a strong foundation to be able to perform critical analysis on game design.

To enable critical analysis, we covered different schools of thought on definitions of a game and play [6][1][14][3][23], game rules [23], games of emergence and progression [14], challenge and conflict and the definition of interactivity [7], rewards and goals [23], and game narrative [15]. Student feedback was invited during lectures, which gave the students opportunities to integrate the knowledge and form their own opinions about these concepts.

However, as the program progressed, we found that the students had some difficulty taking the abstract theories of game design and applying them to their games. This was in part due to the accelerated nature of the program; the students start to design their games by the second week of the program. We were left feeling that we needed a quicker way to introduce design theory that the students could easily relate to and incorporate into their projects.

2.4.2 Practical Design Concepts

In 2010, we attended a tutorial session at the Foundations of Digital Games Conference entitled *The MDA of Bartok* [13]. The session covered the MDA framework, which stands for Mechanics, Dynamics, and Aesthetics. Mechanics refers to the rules of the game, dynamics to the way in which mechanics interact with each other and with the player, and aesthetics is the emotional response of the player to the game—or in what way the game is engaging.

The MDA approach describes how making changes to the mechanics, dynamics, or aesthetics of a game impacts the other two aspects of the game. To demonstrate this, the tutorial focused on the game Bartok (similar to Uno, but played with a regular deck of cards), and how changing the mechanics of Bartok would change the dynamics and aesthetics of the game. After attending the tutorial, we decided that using MDA as a design framework would be an approachable way for the students to dive into game theory and see how it could affect the games they were making. Therefore, in 2010, we switched to teaching students the MDA framework.

We began the class with a slightly modified Bartok activity similar to that presented at FDG 2010. The activity involves having the students play Bartok a number of times while incorporating additional or changed rules. Each rule is added, changed, or subtracted one at a time, so the students are able to see how changing one mechanic can change the dynamic and aesthetic of the game.

This turned out to be a successful introduction to the concepts of game design; the students were able to integrate the theory into their own projects, and they were able to talk about their projects on a much deeper level. We were also able to show how the different aesthetics led to more varied styles of games than the students were used to seeing, and how each could be considered "fun". After introducing MDA, we then began integrating more theoretical design formalisms, tying them back to the framework. For instance, games of emergence and progression are different dynamics that come from changing the mechanics of the game.

While successful, using the MDA framework was not a perfect solution. Many students had difficulty internalizing the concept of dynamics, in part because we found dynamics to be the most challenging to explain. Additionally, there was some initial confusion with the term aesthetics, as many students reverted to its more common definition, and used it to describe the appearance of the game. We feel with some changes to the lecture, such as using more examples and giving the students more practice at using the terms would certainly overcome these challenges.

3. PROJECT DESIGN

One of the primary goals of the COSMOS program is for students to participate in large-scale research or design projects that they can take back to their home institutions to present to their teachers and classmates. In the case of the video game cluster, this is a small, 2D game programmed by students in groups of two or three. Given the short time scale, the primary focus of the programming course from the very first day is on building this game.

3.1 Student Expectations

Each year we have become more emphatic about setting up the expectations of the students. Many of the students arrive thinking that they will be creating 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 an alpha-quality 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. By 2010, we told students at the first day orientation and showed

examples of previous games 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 graphics or physics. Keeping the students focused on a 2D game also discouraged re-implementation of their favorite games, something that was an issue in the first and second year of the program.

3.2 Introduction of a game theme

In the 2010 class, we decided to add a theme for the students to design around. This decision was based on the success of a game design theme in both the annual Global Game Jams[4] as well as the Experimental Gameplay Project[5]. The theme was used to add constraints to the design space which would challenge the students' creativity. We chose the theme of "twilight", with the added constraint that it could not have anything to do with the novels by the same name. While some students addressed the theme on a superficial level (e.g. having the game end when the sun set), others had the theme be a major part of their game. For instance, one group's chose to focus on a more abstract definition of twilight: "a state of ambiguity or obscurity." The game is a puzzle platformer where the player switches between two parallel worlds. Each world had different geometry and physics rules.

We felt that, in general, adding a theme component was a success, but in the future we would choose a different theme. Using a verb instead of a noun could encourage the students to incorporate the theme at more than a story level. Some students had trouble decoupling the theme from the novels, and it was too easy to incorporate the theme as an artistic or minor element. The games that integrated the theme at a mechanic level tended to be more innovative. It is possible that this is somewhat self-selecting (the students who think about the theme more deeply will think about their game designs more), but this can be tested in future iterations of the class.

3.3 Team Selection

For the first three years of the cluster, we allowed students to choose their own teams with the thought that the students would rather work with their friends. However, team selection happened early enough in the program, that students hadn't really learned who they would work well with, or who they would get along with on a month-long scale. Many times, team selection was based more on where the students were sitting than any other factor. This would lead to issues with students working together with drastically different goals for the game, and in some extreme cases, some students were barely talking to each other by the end of the project.

In 2010, we decided to assign teams instead of allowing the students to choose their own teams. Each student sent us an email with three game pitches they would like to work on, and three of their classmates that they would like to work with. To help with team selection, we posted each student's game pitch anonymously to the class wiki. We then created teams based on skill levels, personalities, game pitch compatibility and the student's choices. This turned out to be a long process but was a large improvement in the success of the teams.

By balancing skill levels, we were able to minimize the number of students who felt like they didn't contribute to the game effort. By allowing the students to choose pitches they wanted to work on, it meant that they were able to work on a game they were interested in, even if they weren't with their friends. Overall this was very effective in creating stronger groups who worked well together; some of the teams are still working on their games almost a year later.

3.4 Programming Instruction

Over the four years of the program, we have greatly varied the way we teach students how to program their games. This section discusses the various approaches we've used, from unstructured programming activities to highly structured required 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.

3.4.1 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 disadvantanges, 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 [22]. 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.

3.4.2 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

[4] http://globalgamejam.org/

[5] http://experimentalgameplay.com/

and ownership over their work was clearly very important to them.

3.4.3 Creativity in Small Assignments

In 2010 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 students' understanding of programming concepts by forcing them to read and understand code that wasn't written 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.

3.5 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 [27]. 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, which was quite successful. Building on this success, we added 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.

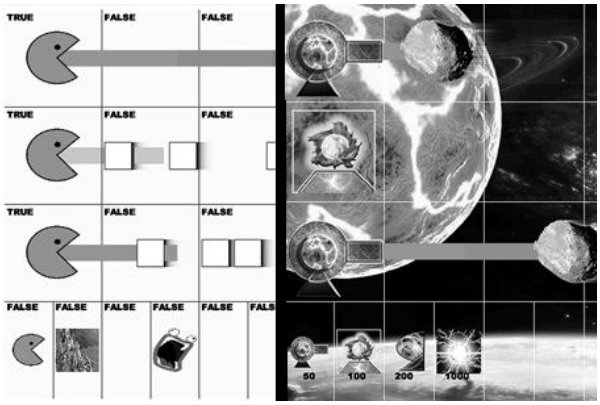

Figure 2. The computational prototype of a column-based tower defense game (left) was used to test tower placement and different shooting styles. After a number of refinements to the prototype, the students went on to complete the game (right).

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, simultaneously exercising the MDA vocabulary. 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.

3.6 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 [9] by instituting a computational prototype assignment. We required students to identify a core mechanic of their game and prototype it (see Figure 2). 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 roguelike 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.

3.7 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 [17], were being left out of programming activities.

In 2010 we required 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 [16], 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.

4. DISCUSSION

Over the last four years of COSMOS, we have made significant changes to our curriculum, all with the goal of enabling student creativity, easing creation of their games, and improving theoretical understanding of games. By iterating on our course's design, we have learned a great deal about what works and doesn't work in teaching game design and programming to students, resulting in a list of concrete recommendations for similar game design programs to follow:

Promoting student creativity within constraints leads to greater student motivation and learning. Students are more motivated to learn when they have creative control over their work. For example, instead of being taught about programming concepts in the abstract, students learn by asking how to solve a particular problem in their project. The introduction of a game theme and requiring individual game pitches before assigning groups is another example. Students are forced to think within the theme constraint, and then later forced to adapt their original idea to meet the constraints imposed by their team mates. This results in initial game design documents that are well thought-out, and discourages the reimplementation of an existing favorite game.

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 game they are making.

Activity-oriented classes help students connect abstract theories to concrete design. Students understand concepts more fully when they are able to apply them shortly after learning them. For example, lecturing about the differences between a game of emergence and a game of progression conveys the knowledge. Following with an exercise to find a game they like, identify whether it is a game of emergence or game of progression, and then change the game to make it the other type of game gives the students opportunities to understand the concept.

Adding instructor control over team dynamics leads to greater learning. While we thought students would do a better job of picking teams than we would, it turned out that the students do not have enough information to pick successful teams. Having the instructor pick teams based on student input is more work but ended up being far more successful.

4.1 Future Work

We have a few ideas for what we would like to change in future iterations of our game design cluster, in both the design course and the programming course.

This was the first year we introduced the MDA framework in the *Structure* class, and some refinement will be beneficial. Some of the critical analysis theory was incorporated, but not to as large of an extent as desired. In future classes, we are looking at integrating more of these theories. 2010 was also the first year that we required a playable physical prototype of the students' games. We felt that this was a success, and we'd like to make them a larger part of the curriculum in the future. There was limited time for the students to try each other's games, and we found this provided valuable feedback. For future classes, we would incorporate more time for the students to play each other's prototypes, and possibly require an additional prototype incorporating the feedback.

Technologies 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 the iterative development of a month-long intensive game design course held as part of the COSMOS summer program at UC Santa Cruz. As the program has matured, we believe that it has gotten much stronger. The students have more fully incorporated the game design theories, they have a stronger knowledge of programming concepts, and their projects have become more innovative. We believe that the lessons we have presented can be used to help shape other game design courses and outreach programs.

5. 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) and Nathan Whitehead (NVIDIA) as former instructors for the Structure of Fun and Technologies of Fun courses, respectively. Finally, we would like to thank the cluster's Teacher Fellow, Takeshi Kaneko, for his insight into teaching methodology, and all of the UCSC COSMOS staff for their support.

6. REFERENCES

1 Abt, Clark C. *Serious Games*. University Press of America, Lanham, MD, 1987.

2 ARMOR GAMES. One Button Bob. (Flash Online Game), http://armorgames.com/play/5286/one-button-bob.

3 Caillois, Roger. *Man, Play, and Games*. Simon and Schuster, Inc., 2001.

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

5 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.

6 Costikyan, Greg. I Have No Words & I Must Design. In *Proceedings of Computer Games and Digital Cultures Conference* (2002).

7 Crawford, Chris. *Chris Crawford on Game Design*. New Riders Publishing, 2003.

8 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.

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

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

11 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.

12 Hunicke, Robin, LeBlanc, Marc, and Zubek, Robert. MDA: A Formal Approach to Game Design and Research. In *Proceedings of the AAAI Challenges in Game AI Workshop* (2004).

13 Hunicke, Robin and Smith, Ben. Tutorial: Applied Game Design - Part 1: The MDA of Bartok. In *Proceedings of the Fifth International Conference on the Foundations of Digital Games (FDG '10)* (Monterey, CA, USA 2009).

14 Juul, Jesper. *Half Real: Video Games between Real Rules and Fictional Worlds*. MIT Press, 2005.

15 Mateas, Michael and Stern, Andrew. Interaction and Narrative. In Zimmerman, Katie Salen and Eric, ed., *The Game Design Reader: A Rules of Play Anthology*. MIT Press, 2005.

16 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.

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

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

19 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).

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

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

22 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).

23 Salen, Katie and Zimmerman, Eric. *Rules of Play: Game Design Fundamentals*. Massachusetts Institute of Technology, 2004.

24 THATGAMECOMPANY. Flower. (PlayStation 3 2009), Sony Computer Entertainment.

25 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).

26 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.

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