Hungry Wolves, Creepy Sheepies: The Gamification of the Programmer's Classroom

David T. Reitter

Abstract The Wolves and Sheep game is an educational, multi-agent computer simulation for students of programming classes. Players move about in a two-dimensional landscape, aiming to reach targets or catch other players. Students write programs that control individual players in the game. The programs then participate in a tournament. Students practice basic programming and algorithmic thinking, object-orientation and the role of interfaces, and are even exposed to basic Artificial Intelligence. Gamification helps motivate students and creates positive, but challenging learning experiences.

Programming can be a tough skill to learn: those who excel at it have started early. Most have spent countless nights doing detective work to figure out why their program doesn't work right. Programming takes a range of abilities beyond robust general computing skills: learning the syntax of an artificial language, the creative use of algebra, and an intuition when to plan ahead and when to try out ideas in practice. Programmers read and write technical documentation, research tricky questions and relate their work to business requirements. Learning these skills is a frustrating mountain ascent, with rocks in the way and many minor falls after which a climber has to get up and try again.

All of this takes a great deal of motivation. A recently popular incentive in areas from crowdsourcing to classroom instruction is gamification. In the following, I will review a tournament of a game run in a programming class that balances teamwork and individual work and addresses important lessons to be learned by novice programmers. The basic idea:

My students play swaps roles for the programmers. They no longer write a computer game to be played by human players—they write small computer programs that act as players in a bigger bio-simulation. The simulated world has players move about a two-dimensional barren landscape with some obstacles and a few meadows. The players are of two kinds: sheep, and wolves. The sheep aim to feed by moving to a meadow. Unfortunately, the wolves need to eat, too.

D. T. Reitter (✉)
College of Information Sciences and Technology, The Pennsylvania State University, University Park, PA 16802, USA
e-mail: reitter@psu.edu

J. M. Carroll (ed.), *Innovative Practices in Teaching Information Sciences and Technology,*
DOI 10.1007/978-3-319-03656-4_3,

The idea of computer programs as independent entities acting in a toy world is not new. The computer science fields of Multi-Agent Systems and Robotics are concerned with such algorithms, and, educationally, the idea can be found as early as the late 1960's, when Seymour Papert, along with colleagues, developed "Logo". In this programming language, the learner would write programs that drew lines on the screen, following simple algorithms. These lines were drawn by agents, which, similar to a spider creating a web, moved around a two-dimensional plane. With this, Papert elicited "body-syntonic reasoning" among his students, who were asked to put themselves into the spider's position. The crucial step is to formulate one's next step—forward, turn, draw, and the like—in an algorithmic way, and ultimately, in the programming language.

The objective of the game we play is not to draw lines on a sheet of paper. The small "agent" programs that the students write, play with and against each other in each round of the game. The next section will describe the game in detail. So far, the reader may imagine a game on a 30×30 board, where each player can make one step at a time in any direction, avoiding obstacles. A players's objective may be to reach a goal location, or to reach another player to "eat" it. The classic video game "PacMan" comes to mind: a student's job would be to write an algorithm that controls the smiling PacMan, or a hungry monster.

I will describe the details of the game later; first, I take the educator's perspective and describe two major considerations that led to the project: team work, and student motivation.

Teaming as a Hard Skill

Basic programming is a skill acquired by individuals. Yet, teamwork in software development is embodied in the design patterns and workflows adopted by the software world. When software developers, project managers and customers work as a team, they use standardized ways of agreeing on expectations and to-do lists for the final project. At a basic level, we strive for an appreciation of such requirements documents among our students, without instigating more corporate red tape. It's simply a way to foster a "Think before you program" attitude.

Collaboration, however, has a very technical side as well. What makes work of several individual programmers connect up is an interface. This is the glue between different pieces of code, which, in turn, are written by different programmers. It defines how data is moved between modules, how code responds with results or error messages, and so on. Modern object-oriented patterns of software development compare well to the organization of large companies: functional responsibilities are delegated to other modules; and modules work semi-independently and cannot "mess with" another module's data. Interfaces represent contracts between individual parties, listing responsibilities and the exact formats used to exchange information.

For a student of programming, this means that they need to read technical documents describing interfaces. They also need to understand how to comply with the contract that is defined by the interfaces. In the class project described here, students write game code that is executed by a larger tournament system, defined by the instructor. The system is documented professionally. When a student ignores details of an interface, it will fail to run.

In short, teamwork is no longer just a soft skill. The education perspective is that software engineering has defined business-level communication procedures and the interface standards for software, so that we need to teach it as a hard skill: explicitly and controlled via assessments.

The Wolves and Sheep game incorporates these notions of "team" in a formal way. Beyond the technical interfaces, there is classical teamwork as well. Students choose to either write a "sheep" or a "wolf" player, and are assigned teams of sheep or of wolves. The "sheep" within assigned groups may work together. This requires, among "sheep" programmers, coordination as well as taking responsibilities for one's teammate's learning. The actual programming work remains an individual effort. The incentive structure discourages free-riding in student teams and the strategic omission of core skills by the students.

Incentives in Learning to Program

The second introductory question we need to address is one of motivation.

Internal motivation can be stimulated: for instance by giving taxing, but enjoyable tasks whose applications to the student's lives and work-lives are apparent. Learners also derive motivation from a feeling of competence (Self-determination theory; Deci and Ryan 1985). Programming is a particularly challenging area to learn due to a steep initial learning curve and the exposure to complex development environments. Finally, it has been suspected that relatedness, a feeling of being connected to others and of being surrounded by a safety net.

External motivation, traditionally provided by grades and degrees, has been systematically eroded by grade inflation, or better grade compression: Students expect grades in the A/B range, usually for "just trying". With high GPA levels overall, this is a fair expectation. Second, as instructors, we are asked to make the customer happy: we are evaluated according to student ratings of "teaching effectiveness", ratings which are useful, but which do correlate with the grades students receive. As a consequence of all of this, grades can no longer provide the same level of external motivation.

So, we need a better incentive structure.

Games can make this happen. They're fun to play, and fun, yet difficult to design. Many of my colleagues have students write games as their first programs—and indeed, my own first programs, which I wrote as a six-year old on a now-primitive machine, were rudimentary, but fun-to-play video games.

That said, a computer game written by a beginner does not come with a measure of quality. What is a good game? An instructor would have to judge that, and there is limited continuous feedback about the quality. How does a student know they're on the right track?

Each game of Wolves and Sheep is executed automatically in a tournament system. It systematically runs thousands of such games, combining all sheep teams with all individual wolves and creating a tournament. After hundreds, perhaps thousands of automatic and blazingly fast runs, the tournament produces a ranking between the agents.

During class-time, we run the tournaments and discuss results. Outside of class, students can submit their programs at any time via a web site, and a central server executes them, playing about one full tournament every hour. The scores are visible to everyone. This way, students can iteratively improve their algorithms. They can try out new algorithms and focus on what works well.

Gamification creates competition in the classroom. Winning the game involves not just doing well individually—it means beating some classmates. Psychologically, this creates motivation, although my hope is that the game is intellectual stimulation and a learning exercise more than anything, as students's agents serve as proxies in the actual gameplay.

Of course, how well a student reacts to this form of competition is a personal matter. Research has pointed out gender differences, with a reduced motivation to compete among women (Niederle and Vesterlund 2007). However, more recent work points out that these attitudes are only initial. Economist Joseph Price summarizes his work: "We found gender inequities disappear very quickly if you do more than one round of a competition. In the time since we first conducted our experiment, there have been other studies that have shown similar things. One of the reasons girls don't do well in competitive settings is they don't think they're as good as boys—but they really are" (cf., Cotton et al. 2013).[1] Anecdotally, women do very well in my class, although they—like some of their male peers—point out that it creates a high-pressure environment. So, I would emphasize that games do not alleviate pressure. However, they shift its source away from an instructor-imposed grading scheme. I aim to have my students feel they "want to accomplish the learning goals", rather than "they need to finish the computer program".

Wolves and Sheep

The game my student's agents play is called Wolves and Sheep.

As the name suggests, it is a non-serious computer simulation with simple rules inspired by biology. Two species share a piece of land: wolves, and sheep. All players start each round of the game at assigned positions. Sheep wander around in search for green grass. Wolves run across the land to hunt sheep. If a sheep arrives at

[1] http://www.edmediacommons.org/forum/topics/five-questions-for-economist-joseph-price-on-rethinking-gender-di—Interview posted in March 2013.

a pasture, it gets to exit the round, gaining a point. If a wolf finds an unlucky sheep, it gets to eat it. The hungry wolf continues the game after a while, but the sheep exits the current round.

Wolves and sheep are not controlled by students directly. This being a programming class, students are tasked with writing computer programs that steer the wolves and sheep around the countryside. Each program controls either a sheep, or a wolf, and because there are four sheep and one wolf in each round, the players programmed by five students compete in a single round of the game. The animals move along grid coordinates, much like those used in North-American cities or on maps. They can move North, South, East or West—or diagonally, but never further than a prescribed distance at a time. Wolves are a little faster than sheep, but how fast exactly they can be is information only available late during the development phase. The small computer programs do not move randomly: obviously, sheep have to jump towards pastures, avoid obstacles, and, of course, run away from the wolf. Wolves need to hunt for sheep, that is, they are chasing moving targets.

The two-dimensional worlds these animals live in come in different scenarios: sometimes, obstacles are arranged in a way that requires some navigation. Green pastures and start locations of the players are not always in the same place. Students are given about 10 practice scenarios, but the tournament is run in earnest with additional, new ones. Such caveats force students to think about flexible strategies and implementations, which is an important real-world requirement. If their programs assume that obstacles or pastures are always in the same place, it will lose.

Each scenario starts with four sheep and one wolf at predefined or random locations. All players then take turns moving about; sheep naturally try to get to a green pasture as quickly as possible. The wolf attempts to get to and eat as many sheep as possible before they reach the pasture. The game ends when each sheep is dead or grazing happily, or when a timeout is reached. Figure 1 shows two example scenarios with a selection of players.

Writing a computer program that does this well takes much planning, but also requires the tools of the trade. To write a sheep or a wolf program, students have to conform to the documented interfaces. These interfaces describe how a program may find out where the other players are in the game, where the obstacles are, and so on. They determine how the program communicates to the tournament system where the sheep or wolf should move next (one step diagonally North/West, for example). In a realistic computer system, software interfaces are used millions of times every minute: when a user highlights a sentence with the mouse, the operating system tells the program (such as “MS Word”) via an interface where the mouse click occurred, and/or what was highlighted. Similarly, many layers of interfaces describe how a web server may provide a website to a user’s browsers. The wolf/sheep interfaces are a miniature version of this, and they tell an important life lesson to the early-stage programmer: you must comply with the rules.

A sheep program interacts with the tournament system in various ways. The tournament code, of course, organizes about 30 different wolf and sheep programs supplied by the students. In each round, it keeps track of where the four sheep and the wolf is. When the time comes for a sheep to make a move, the tournament system

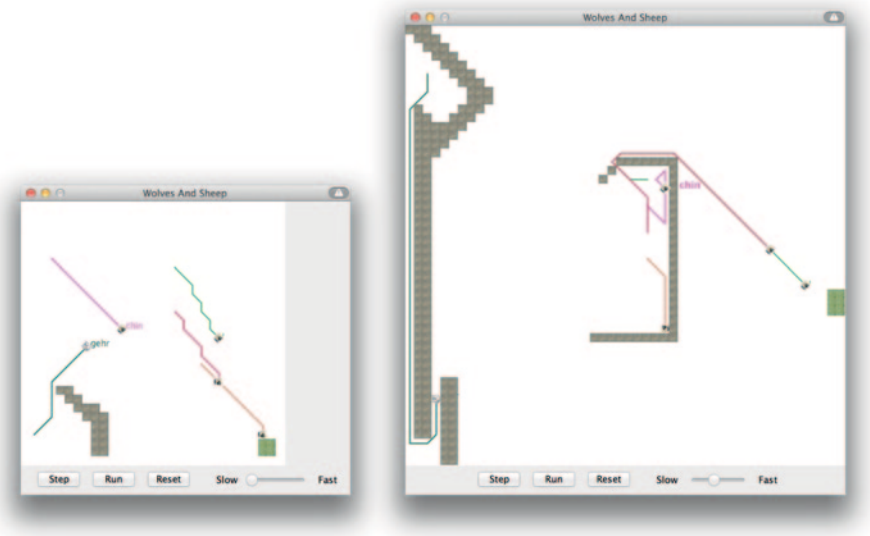

Fig. 1 Two of the 8 scenarios given to students to practice, along with the trajectories of the sheep and wolf players during the game, starting from their initial positions. The left-most player shown in each diagram is the wolf. In the scenario on the left, the wolf just has to circumnavigate a simple obstacle, while on the right, obstacles require more sophisticated path planning algorithms for sheep and wolf. There, two of the sheep do not have such algorithms, while the two others and the wolf clearly demonstrate some planning ability. The graphical environment lets programmers pause and slow down the simulation

asks the sheep program via the interface: what's your move? Now, the student's program runs. Via the interface, it asks the tournament: Where are the other sheep on the game board, and where is the wolf? The tournament system replies truthfully. Next, it might ask, Where are the pastures? There is no function to automatically find out which one of the pastures is closest, so the student has to write an algorithm to do that. She also needs to figure out how to move towards the pasture. For instance, if the pasture is at the top right, and the sheep is in a center left location on the game board, it could move two steps to the right, and one up. Because that would violate the game rules (the sheep can't run that fast), it will have to move one step up this time, and one to the right for the next two moves. Finally, the student's sheep program returns control to the central tournament system, with the information: I'll move one step up! The tournament system checks that the sheep is within the boundaries of the game board, and that it is not about to step on an obstacle. If it reaches a pasture, the tournament awards a point to the student. If it collides with wolf, disaster ensues for the sheep.

Introduction in the Classroom

Students sometimes start our second programming course with very limited computer experience. The formal prerequisites notwithstanding, foundational concepts such as variables or functions are forgotten, and even high-school algebra no longer feels intuitive. Expecting previously taught skills is the right message to send, but pragmatism dictates that we get our class up to speed early on. The first weeks of the course are dedicated to a textbook-supported re-cap, taught with standard materials, multiple short quizzes, computer-based tutoring and even an old-fashioned requirement to memorize the exact syntax of the 6-line Hello World program.

Before we get to the Wolves and Sheep game, students complete a mid-term mini project, as individuals. There, they implement a simple Tic Tac Toe game (a common educational task in programming courses), and, for full credit, they can choose to attempt to write a simple Tic Tac Toe player for boards larger than just a 3-by-3 grid. The results of the exercise demonstrate the range of abilities in my classroom. The best students attempt to implement advanced Artificial-Intelligence algorithms, while others struggle with functions displaying the game board on the screen. All of them are programming beginners, and the expected performance is a simple computer player that does not make basic mistakes. The exercise allows all students to understand the idea of writing a computer player for a game. We do run a tournament that pitches different computer players against each other. Students peer-review each other's code.

Early on, I provide students with some sparring partners: a selection of Wolf and Sheep agents programmed by students of previous classes, as well as my own. These player programs will run, but are encrypted to prevent casual reading as a source of inspiration.

One Sheep I wrote myself in order to demonstrate the value of simple heuristics: basic algorithms that aren't perfect, but that can go a long way in solving the main problem. To my surprise, one of these programs won the competition in one of the weaker classes. A student, who wrote a very complex and advanced Wolf player, commented that my program did well because it was well executed: it got small details right, even though it did not come up with a perfect big-picture solution. For instance, Wolf programs may mis-navigate. They race towards a sheep, but at the last step, they hop over the sheep rather than actually hitting it. Human (and animal) common sense prevents this, but a computer program lacks common sense. Paying attention to such details is an important lesson that my class learned that day.

Team Structure

At the beginning of the project, students choose to either write a wolf or a sheep program. Four students are grouped into wolf or sheep teams (at their choosing). The students in these teams are asked to help each other with the programming task.

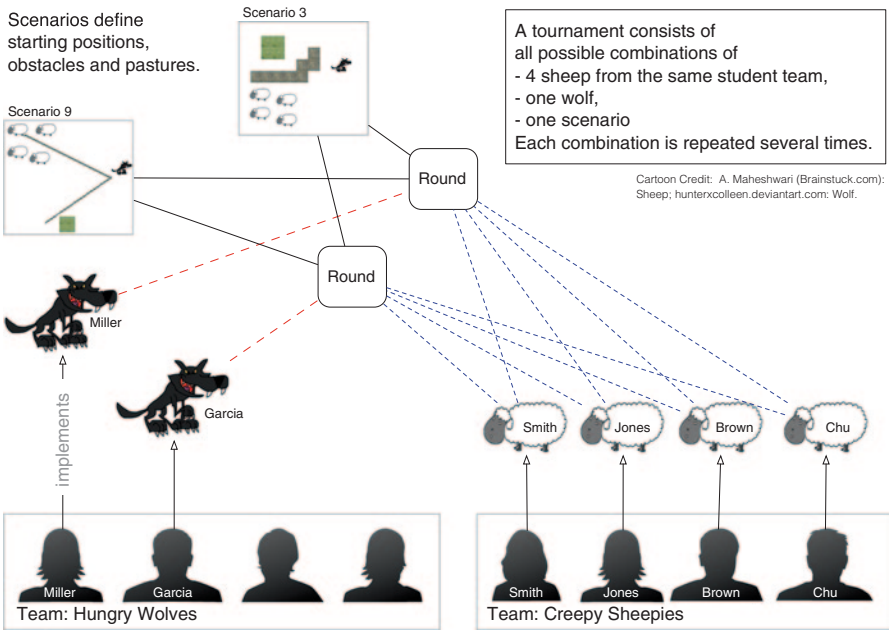

Fig. 2 Student programmers supply programs that compete in many rounds of the tournament. The final tournament includes many scenarios and repeats each combination. Each student's program typically plays 25,000 times before the results are tallied up. The tournament can be run in about 30 min with 25 participating student programs

Figure 2 describes the grouping within the game: a lone wolf against four sheep. The four sheep in a round always stem from a single student team. While each student is to develop their own sheep, I encourage them to work together.

Conversely, the sheep programs by each of the sheep team members can exchange information during the game to outsmart the wolf: for instance, they can agree that one of the sheep is to lure the wolf away from the rest. If the sacrificial lamb is chosen at random and the round is repeated many times, no single student will be at a disadvantage. Instead, the overall team improves its score. In an untested (future) modification of the rules, I will allow sheep to overwhelm a wolf if and only if they act as a team: all four sheep can surround and eliminate the wolf. Unlike sheep, wolves act as loners in the game.

To incentivize team work, the final grade assigned to each students consists equally of (a) the performance of the sheep/wolf in the final tournament, (b) the performance of the student's team, and (c) the quality of the written code. The tournament gives each wolf program a point for each sheep it eats in any round; a sheep gets a point when it reaches a pasture. Of course, performance depends on the particular opponents in the tournament. To grade fairly, other wolf and sheep program (from previous classes, for instances) may be thrown into the mix and used to set standards. Performances of wolves and sheep are not compared to each other.

Continuous Improvement

The game system demonstrates the value of regular testing, as it would be used in software projects. Early on, students are encouraged to submit a simple Wolf or Sheep agent to the central tournament system. In early classes, I asked a teaching assistant to receive and check the submissions. This proved too time-consuming: students typically went through several iterations until they understood that they had to comply with the interface specification that defines how a Sheep or Wolf agent talks to the tournament system that runs the game. Now we have a website that takes in the program and checks it automatically as far as feasible.

Students may submit their program to the server at any time. The last available submission is used in regular tournament runs, and every hour we provide a new ranking via the website. This process takes place automatically. Students can improve their Wolves and Sheep at each step. If a change makes performance worse, they can backtrack. If their competitors improve and their ranking declines, they feel motivated to step up their game.

Listening to computer security specialists and computer educators, I took steps to protect the tournament code against cheating and hacking. Actual attacks turned out to be rare and benign. Typically, I reward students who find ways to circumvent interface limitations: they will benefit greatly from the learning experience.

What was a frequent problem, though, was student-supplied code that behaved in unexpected ways: it was slow or did not finish the job, crashed, or failed to comply with game rules (a sheep moving two steps at a time, for instance). The tournament system has to be robust, and also punish the player's program in the ranking if that happens. The learning effect on students is tremendous: they have to pay attention to detail and make sure everything works. Since students are given the actual game code, they will test their program on their own computer before submitting it to run in the central tournament. It is important to provide a system that gives timely feedback if students write unacceptable code.

During this phase of the class, I become more of a coach than a rulemeister—game rules are enforced by the computer system (that I wrote). As a coach, my job was to help each student out to the best of my ability, while keeping proprietary ideas secret.

The tournament system is capable of running many thousands of rounds of the game in the background, but it can also visualize a single round with a few sheep and a wolf on a computer screen (user interaction and visualization using Pluss 2013). This lets the programmer trace the player's steps and figure out the dynamics. At the end of the semester, this is what we do in class, to much laughter and excitement. At that time, the behavior or the bots can still be a surprise, when scenarios are introduced that the students haven't seen during development.

Throughout the project, I found that it is important for students to make mistakes. The Wolves & Sheep project provides many opportunities for that. Careful automated checks at many stages will present the learner with error messages and a chance to fix the problem. In learning to program, every mistake is a learning opportunity. I convey to my students that even very experienced programmers make mistakes: even wide-spread software has its faults.

Critical Reflection

Without a careful, statistically meaningful evaluation of learning gain I won't claim that competitive gamification is a better way to teach programming. It is also not a panacea for other student woes: it does not relieve students from the need to understand the basics, and the theoretical model behind a programming language. Those who start the project late still feel pressured to perform. Students still need to spend the time to practice. The game, like many of my programming assignments, is open-ended in certain ways and requires a great deal of learner autonomy (Holec 1981). The course I give embraces active learning, as it has students do some tasks autonomously and asks them to reflect on their progress through the semester.

Games can be tremendously motivating: as one student wrote to me, "some nights ended up very late because of them." What reinforces this is that grades are tied to the outcome. However, fair grading requires care. I tend to assign grades based on scores rather than ranks, which are more susceptible to small variations and the performance of in-class-room competitors. As pointed out before, quality of the code is also judged directly, and the project is only one aspect of the final grade. The Wolves & Sheep game assesses students on their overall ability to write computer programs, including any experience they brought with them to class. Programming abilities are cumulative, and grading based on the just the student's learning in a specific class is difficult to achieve. The Wolves & Sheep game comes with a great deal of flexibility (in the sense laid out in the introduction to this volume). Some students implement complex, very smart wolves, by using existing techniques from Artificial Intelligence. Others come up with their own, simpler solutions. Many of them arrive at a sense of achievement. The game scales well as an educational tool.

Unlike in a quantitative study, these classroom experiments do not yield generalizable results.

I benefitted from daily feedback given to me by the students (via an electronic "log book"). "This class," wrote one student, "thankfully strayed from 'academic coding' and instead taught 'real' coding. Getting exposure to APIs and different algorithms was invaluable". Admittedly, the game made AI a focus. A student sees that as a shortcoming, as the game is "based on more of the quality of the artificial intelligence implemented instead of the implementation of object-oriented programming." Other students struggle with the transition from the programming concepts taught via examples to the application of those concepts embodied in the Geo Game. However, perspectives differ. Writes another student: "The projects were awesome. Also, the best policy to have [is] learning not just about Java, but how to program."

Acknowledgements I would like to acknowledge the, despite efforts, untraceable source of the idea of a similar educational game that has been used in programming classes at Carnegie Mellon University. I thank Martina Vasil for her extensive comments, and my students at Penn State for their valuable feedback during the development of the game and in past classes.

References

Cotton, C., McIntyre, F., & Price, J. (2013). Gender differences in repeated competition. *Journal of Economic Behavior & Organization, 86,* 52–66.

Deci, E. L., & Ryan, R. M. (1985). *Self-determination*. Wiley.

Holec, H. (1981). *Autonomy and foreign language learning*. Oxford: Pergamon.

Niederle, M., & Vesterlund, L. (2007). Do women shy away from competition? Do men compete too much? *The Quarterly Journal of Economics, 122*(3), 1067–1101.

Pluss, A. (2013). The JGameGrid Framework: An education oriented java package for developing computer games. http://www.aplu.ch/jgamegrid/. Accessed Oct 2013.