Teaching Artificial Intelligence Playfully*

Mike Zyda and Sven Koenig
University of Southern California (USC)
Computer Science Department
Los Angeles, California (USA)
{zyda,skoenig}@usc.edu

Abstract

In this paper, we report on the efforts at the University of Southern California to teach computer science and artificial intelligence with games because games motivate students, which we believe increases enrollment and retention and helps us to educate better computer scientists. The Department of Computer Science is now in its second year of operating its Bachelor's Program in Computer Science (Games), which provides students with all the necessary computer science knowledge and skills for working anywhere in industry or pursuing advanced degrees but also enables them to be immediately productive in the game development industry. It consists of regular computer science classes, game engineering classes, game design classes, game cross-disciplinary classes and a final game project. The Introduction to Artificial Intelligence class is a regular computer science class that is part of the curriculum. We are now converting the class to use games as a motivating topic in lectures and as the domain for projects. We describe both the new bachelor's program and some of our current efforts to teach the Introduction to Artificial Intelligence class with games.

Introduction

Games allow universities to teach computer science hands on and motivate students because playing games is fun (Cliburn 2006). In addition, games allow students to debug their code easily since the code can be tested by playing the games, with visual results. It is therefore not surprising that many universities investigate how to use games to teach computer science, as evidenced by papers in the Game Development in Computer Science Education Conference and the Frontiers in Education Conference. The Department of Computer Science at the University of Southern California (USC) is now in its second year of operating its Bachelor's Program in Computer Science (Games) and Master's Program in Computer Sciences (Game Development) (Zyda 2006). An important aspect of the bachelor's program is to motivate students to learn computer science, and thus to boost student enrollment and retention, which is important since the US needs to double the number of science and technology graduates by 2015 according to the July 2005 report of the TAP Forum (Business Roundtable 2005). The bachelor's program is designed to make the students better computer scientists as well as improve skills typically neglected in computer science programs, such as being creative and working in teams with very different expertise. It provides students with all the necessary computer science knowledge and skills for working anywhere in industry or pursuing advanced degrees but also enables them to be immediately productive in the game development industry since they are not only strong programmers and system developers but also understand the game development process well. It consists of 37 units of computer science classes and 42 units of game development classes. It was created not by watering down the regular bachelor's program in computer science but rather by replacing elective slots with game development classes, making it look very much like a double major. As such, the bachelor's program consists of regular computer science classes, game engineering classes, game design classes, game cross-disciplinary classes and a final game project. Some of the regular computer science classes, including CSCI 460 (Introduction to Artificial Intelligence), are now being converted to use games as a motivating topic in lectures and as the domain for projects. In the following, we describe both the bachelor's program and our current efforts to teach CSCI 460 (Introduction to Artificial Intelligence) with games.

USC

USC was able to build on several strengths to build the bachelor's program, including its top-ranked engineering and cinema schools. Some researchers at USC's Information Sciences Institute (ISI) work on serious games, that is, games whose primary purpose is training and education rather than entertainment. Researchers at USC's Institute for Creative Technology (ICT) develop engaging immersive technologies for learning and training in cooperation with the entertainment industry and the army. For example, the research of Mike van Lent, a researcher at ICT, on explainable artificial intelligence was integrated into the Microsoft Xbox title "Full Spectrum Warrior," the top selling Xbox game in June 2004. Researchers at the Interactive Media

*Our research was partly supported by a grant from the Fund for Innovative Undergraduate Teaching of the University of Southern California.

- **Core Requirements** (43 units)
  - **General Education** (20 units)
  - **Science** (4 units)
    - PHYS 151LG (4 units): Fundamentals of Physics I
  - **Writing** (7 units)
    - WRIT 140 (4 units): Writing and Critical Reasoning
    - WRIT 340 (3 units): Advanced Writing
  - **Mathematics** (11-12 units)
    - MATH 125 (4 units): Calculus I
    - MATH 126 (4 units): Calculus II
    - MATH 225 (4 units): Linear Algebra and Differential Equations
    - EE 241 (3 units): Applied Linear Algebra (alternative to MATH 225)
    - optional but highly recommended classes
      - MATH 226 (4 units): Calculus III
      - EE 364 (3 units): Introduction to Probability and Statistics
- **Computer Science** (37 units)
  - **Programming and Software Development** (16 units)
    - CSCI 101 (3 units): Fundamentals of Computer Programming
    - CSCI 102 (4 units): Data Structures
    - CSCI 105 (2 units): Object-Oriented Programming (C++ Version)
    - CSCI 201 (4 units): Principles of Software Development (C++ Version)
    - CSCI 377 (3 units): Introduction to Software Engineering
  - **Theory** (6 units)
    - CSCI 271 (3 units): Discrete Methods in Computer Science
    - CSCI 303 (3 units): Design and Analysis of Algorithms
  - **Hardware and Systems** (9 units)
    - CSCI 402 (3 units): Operating Systems
    - EE 450 (3 units): Introduction to Computer Networks
    - CSCI/EE 352 (3 units): Computer Organization and Architecture
  - **Autonomy and Immersion** (6 units)
    - CSCI 460 (3 units): Artificial Intelligence
    - CSCI 480 (3 units): Computer Graphics
- **Game Development** (42 units)
  - **Game Engineering** (11 units)
    - CSCI/ITP 380 (4 units): Videogame Programming
    - CSCI/EE 452 (3 units): Game Hardware Architectures
    - CSCI 487/ITP 485 (4 units): Programming Game Engines
  - **Game Design** (8 units)
    - CTIN 488 (4 units): Game Design Workshop
    - CTIN 484 (2 units): Intermediate Game Development
    - CTIN 489 (2 units): Intermediate Game Design Workshop
  - **Game Cross-Disciplinary** (17 units)
    - CSCI 180 (3 units): Survey of Digital Games and their Technologies
    - CSCI 280/ITP 280x (4 units): Videogame Production
    - CSCI 281 (3 units): Pipelines for Games and Interactives
    - CTAN 443 (2 units): 3D Animation and Character Animation
    - CTAN 452 (2 units): Introduction to Computer Animation
    - CSCI 486 (3 units): Serious Games Development
  - **Final Project** (6 units)
    - CSCI 491a (4 units): Final Game Projects
    - CSCI 491b (2 units): Advanced Game Projects
- **Technical Electives** (6 units)

Figure 1: Bachelor's Program in Computer Science (Games)

Division of USC's School of Cinematic Arts study both commercial and internally developed games with the goal of creating a body of knowledge about players, the games they play, and how they play them to design games that exhibit new and better play mechanics and create satisfying new social interactions. For example, the game "flOw" was part of Jenova Chen's thesis research at the Interactive Media Division and is now commercially available on PlayStation 3 game consoles. Other award-winning games include "Cloud" and "The Misadventures of P.B. Winterbottom." Bing Gordon, the Executive Vice President and Chief Creative Officer of Electronic Arts since March 1998, teaches classes in the Interactive Media Division. In February 2007, USC established its USC Games Institute to create a science of games (Zyda 2007), unify and represent USC game research on and off campus, and reach out to the rapidly growing number of video game companies in Los Angeles and elsewhere.

Bachelor's Program

The bachelor's program was created in large part by the first author of this paper, a principal force behind the creation of the online game America's Army at the MOVES Institute of the Naval Postgraduate School (Zyda *et al.* 2005) and now director of USC's GamePipe Laboratory, with the help of many faculty members (including the second author of this paper) and administrators at USC. Figure 1 gives an overview of its structure, and the following sections describe it in more detail, using a shortened version of (Zyda, Lacour, & Swain 2008). We use the following abbreviations for classes from different departments: CSCI = Computer Science, CTAN = Animation Program of the School of Cinematic Arts, CTIN = Interactive Media Program of the School of Cinematic Arts, EE = Electrical Engineering, ITP = Information Technology Program, MATH = Mathematics, PHYS = Physics, and WRIT = Writing.

Regular Computer Science Classes

USC uses games in regular computer science classes early on, both as motivating topic in lectures and as domain for projects. For example, CSCI 101 (Fundamentals of Computer Programming) teaches C++ using a semester-long small game project where the students extend a simple ping-pong game. A special section of CSCI 201 (Principles of Software Development) then builds a simple 2D networked game in C++ with the entire class participating as a group. As another example, CSCI/EE 352 (Computer Organization and Architecture) teaches computer organization and architecture from a computational perspective, including low-level programming. CSCI/EE 452 (Game Hardware Architectures) then teaches programming the Playstation 3, Xbox 360 and GPU hardware as a case study.

Game Design Classes

Game engineers should understand game design. Game design classes teach students how to design games on paper, stimulating their creativity. They introduce a friendly element of competition into the classroom, namely to design the most fun game. USC uses a three-class sequence of game design classes run by USC's School of Cinematic Arts that uses a process called play-centric design (Fullerton 2006), namely CTIN 488 (Game Design Workshop), CTIN 484 (Intermediate Game Development) and CTIN 489 (Intermediate Game Design Workshop).

Game Development Classes

Game engineering classes teach students how games work and how they can code their games so that they can play them afterwards. USC created a two-class sequence of game engine programming, namely CSCI/ITP 380 (Video Game Programming) and CSCI 487/ITP 485 (Programming Game Engines). The idea behind this sequence is to use a high-level toolkit (Microsoft XNA Game Studio Express) in the first class and an actual game engine (built on top of the Ogre3D rendering engine) in the second class.

Game Cross-Disciplinary Classes

Some game classes are cross-disciplinary classes that bring together students with different backgrounds, including engineering, interactive media, animation, fine arts and music, since game design and development requires a variety of knowledge and skills. The fine arts students come from a 2D and 3D game art and design minor. CSCI 180 (Survey of Digital Games and their Technologies) gives a comprehensive survey of the history of videogame technology and game design by combining historical lectures, discussions, and student research presentations with a lab that enables students to play consoles and game emulations from the early 1970s to today. CSCI 280/ITP 280x (Videogame Production) gives an introduction to game development via a combination of lectures, motivational overview talks by guest speakers from the game industry and a lab in which students build individual games using GameMaker. CSCI 281 (Pipelines for Games and Interactives) teaches students how to build 3D models and animations and manage game assets. CTAN 443 (3D Animation and Character Animation) teaches students how to develop and animate game characters. CTAN 452 (Introduction to Computer Animation) teaches students the fundamentals of animation using commercial tools. CSCI 486 (Serious Games Development) teaches students how to build games whose primary purpose is learning and teaches them how to assess those games for their ability to provide that learning. All the projects of this class are built for real clients in areas such as immunology, Russian political history, fish biology, disaster command, fire-fighting command, airplane assembly and financial management.

Final Project Classes

Final game project classes give students a more holistic view of computer science and teach or reinforce a variety of skills, including computational thinking skills, software engineering skills, programming skills, game development skills, artistic skills, problem-solving skills and teamwork skills (such as collaboration and communication skills) in teams with very different expertise, both to develop an engaging game and to code it. CSCI 491a (Final Game Projects) is a game cross-disciplinary class that brings together all of the skills learned by the students to develop a game in large cross-disciplinary teams. There is a production pipeline: The teams receive game designs from CTIN 484 (Intermediate Game Development) and CTIN 489 (Intermediate Game Design Workshop), that were taught in the previous semester, and assets from CSCI 281 (Pipelines for Games and Interactives) and CSCI 443 (3D Animation and Character Animation) and game-engine technology from CSCI 522 (Game Engine Development), which are taught concurrently. 491b (Advanced Game Projects) is used to further develop these games and get them ready for competition. The students demonstrate these games in a joint demo day at the end of each semester to representatives from the game and computing industry, including Sony, Electronic Arts, Activision, THQ Interactive, Digital Domain, Seven Studios, Tactical Language, National Geographic, Dassault, iSportGames, Lockheed Martin, Disney, Emsense, Harmonix Music, Kotrala, Motorola, Applied Minds, Northrup Grumman, Big Stage, Sandia National Laboratories, Konami, Lucas Arts, and Pandemic.

Enrollment, Retention and Job Opportunities

In Fall 2006, there were only 223 applicants to the Bachelor's Program in Computer Science. In Fall 2007, after the introduction of the Bachelor's Program in Computer Science (Games), there were 219 applicants to the Bachelor's Program in Computer Science and 164 applicants to the Bachelor's Program in Computer Science (Games). Thus, the application rate increased substantially, which doubled the enrollment figures and thus achieved one of the intended purposes. It is still too early to evaluate whether the retention rate increased as well. Currently, the undergraduate students in computer science are split about evenly into both programs, with no statistically significant difference in their GPA averages. USC managed to put 30 students into internships and jobs in Spring 2007 and 50 students in Spring 2008 at game and computing industries such as Electronic Arts, THQ Interactive, Disney, Activision, Blizzard, Apple, Emsense, Big Huge Games, 7 Studios and Applied Minds.

Artificial Intelligence

CSCI 460 (Artificial Intelligence) is a regular computer science class that is part of the bachelor's degree and regularly taught by the second author of this paper. We are now modifying the class to use games as a motivating topic in lectures and as the domain for all projects, which makes sense since artificial intelligence becomes more and more important for game development, now that games use graphics libraries that produce stunning graphics and thus no longer gain much of a competitive advantage via their graphics capabilities. Many games need path-planning capabilities and thus use search methods. Some games already use machine learning or planning methods. For example, "Black and White" uses a combination of inductive learning of decision trees and reinforcement learning with neural networks, and "F.E.A.R" uses goal-oriented action planning. Thus, games can be used to illustrate many areas of artificial intelligence and, furthermore, provide research challenges for artificial intelligence (Buro 2003). It was tempting for us to use a publicly available game engine throughout the class and then ask the students to perform several projects in it, such as a search project to plan the paths of the game characters and a machine-learning project to make them adapt to the behavior of their opponents. However, the students in the Bachelor's Program in Computer Science (Games) already have plenty of exposure to game engines while the students in the Bachelor's Program in Computer Science do not need the additional overhead. To put all students on the same footing and allow them to concentrate on the material taught in the class, we decided to go with several small projects that do not need a large code base. We now describe two different projects that we are currently developing for the undergraduate and graduate introductions to artificial intelligence (two different classes with similar syllabi), a step-by-step one for search and a more open-ended one for machine learning. They are challenging projects that require ambitious, motivated and smart students. Some might argue that they are more suitable for graduate students than undergraduate students, which we still need to evaluate.

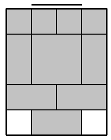

Figure 2: Sliding-Block Puzzle Solvable with 10 Moves

Search

We are developing a step-by-step project for search. Heuristic search and, in particular, A* are among the important single-agent search techniques and thus good candidates for a project in artificial intelligence. An obvious choice for a project is to use A* to solve the eight puzzle or similar sliding-tile puzzles (Hordern 1986), perhaps with different heuristics. However, information on how to solve sliding-tile puzzles with A* is covered in almost all textbooks. We therefore used to use sliding-block puzzles, where one has to move blocks of different sizes and shapes in a rectangular enclosure to move a given block to a given goal position. Sliding-block puzzles are only seldomly covered in textbooks.[1] One can ask the students to design informed consistent heuristics for them and then solve them by coding and running A* with these heuristics. There exist difficult instances of sliding-block puzzles since solving them is PSPACE-complete in general (Hearn 2004), which makes A* quickly run out of memory. Thus, care must be taken to select relatively easy instances, such as instances with small enclosures and short solutions. Figure 2 shows such an instance, where the large block has to be moved so that it could be moved out of the exit on top. However, both sliding-tile puzzles and sliding-block puzzles are more puzzle-like than game-like.

We are therefore developing a project where the students use a generalization of A* (that we call Adaptive A*) to move game characters in initially unknown gridworlds to a given goal cell. The students need to code A* and Adaptive A* and then develop a good understanding of A* and heuristics to answer questions that are not yet covered in textbooks. We give them a description of Adaptive A* and its application to path planning in initially unknown gridworlds. Figure 5 gives a shortened description that closely follows (Koenig & Likhachev 2005a). The search project is versatile since it allows for theoretical questions and implementations:

- One can ask the students to code A* in a way so that it searches forward and breaks ties among cells with the same f-value in favor of a cell with either the smallest g-value or the largest g-value and then explain the difference in the number of expanded cells or in runtime in randomly generated mazes that are generated with a terrain generator that they are provided with (easy).
- One can ask the students to code A* both in a way so that it searches forward from the current cell of the game character to the given goal cell and in a way so that it searches backward from the given goal cell to the current cell of the game character and then explain the large difference in the number of expanded cells or in runtime (moderately difficult).
- One can ask the students why Adaptive A* leaves initially consistent heuristics consistent and makes them more and more informed (to be precise: no less informed) over time, which can be shown with two simple proofs (moderately difficult).
- One can ask the students to code both Adaptive A* and A* in a way so that they search forward from the current cell of the game character to the given goal cell and then measure the difference in the number of expanded cells or in runtime (easy).
- One can ask the students how to generalize Adaptive A* to the case where the given goal cell moves, for example, because the game character is trying to catch a hostile game character or because the game character is trying to catch up with a friendly game character (difficult).
- One can ask the students to explain the behavior of Adaptive A* if the action costs of actions can increase as well as decrease, which can happen in real-time computer games (easy).
- One can ask the students how to generalize Adaptive A* so that it continues to find cost-minimal paths if the action costs of actions can increase as well as decrease (difficult). Alternatively, one can ask them to read up on D* Lite (Koenig & Likhachev 2005b) (which has this property and is thus the path-planning method of choice for real-time computer games), code both Adaptive A* and D* Lite, and measure the difference in runtime (difficult).

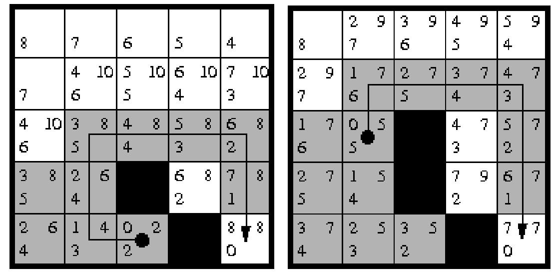

Figure 3: Forward A* Searches

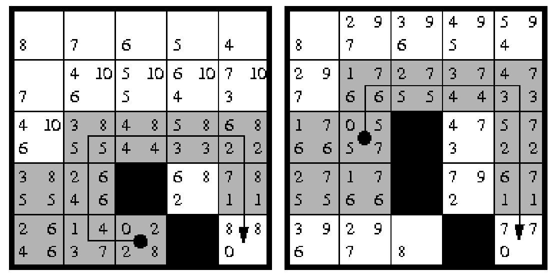

Figure 4: Adaptive A*

Overall, the search project is not easy since it requires students to develop a good understanding of A* and heuristics. (Solutions are available from the second author of this paper on request.) We learned that the students need a longer example to understand exactly how the game characters are supposed to move, when they observe additional blocked cells and when they are supposed to search. We have therefore recently created a longer version of the project that

[1]Examples of sliding-block puzzles can be found at www.puzzleworld.org/SlidingBlockPuzzles/4x5.htm and www.pro.or.jp/ fuji/java/puzzle/slide/V1.0/fuji.index.html.

We explore the following way of making A* more efficient when it solves a series of similar search problems, resulting in a version of A* that we will call Adaptive A*. The task of Adaptive A* is to repeatedly find cost-minimal paths to a given goal state in a given state space with positive action costs. The searches can differ in their start states. Also, the action costs of an arbitrary number of actions can increase between searches by arbitrary amounts. Adaptive A* uses informed h-values to focus its searches. The initial h-values are provided by the user and must be consistent for the initial action costs. Adaptive A* updates its h-values after each search to make them more informed and focus its searches even better. An iteration of Adaptive A* proceeds as follows: It first updates the action costs, if necessary, to reflect any increases in action costs. It then runs a forward A* search to find a cost-minimal path from the start state to the goal state. Assume that the search determined that the cost of the cost-minimal path is $g*$. Let *CLOSED* be the set of states that were expanded during the search. Then, Adaptive A* sets $h[s] := g* - g[s]$ for all states $s \in CLOSED$, where $g[s]$ is the g-value and $h[s]$ is the h-value of state s after the search. It then starts a new iteration. One can prove that the h-values of the same state are monotonically nondecreasing over time and thus indeed become more informed. One can also prove that the h-values remain consistent and Adaptive A* thus continues to find cost-minimal paths over time without having to re-expand states during the same search. This principle was first described in (Holte *et al.* 1996) and later rediscovered in (Koenig & Likhachev 2005a). Now consider characters in real-time computer games such as Total Annihilation or Warcraft. The game characters often do not know the terrain in advance but automatically observe it within a certain range around them and then remember it for future use. To make the game characters easy to control, the users can click on some position in known or unknown terrain and the game characters then move autonomously to this position. We discretize the terrain into cells that are either blocked or unblocked and assume for simplicity that the game characters can only move in the four main compass directions with unit action costs and thus operate on four-connected gridworlds. As heuristic estimate of the distance of two cells we use the consistent Manhattan distance. The game characters initially do not know which cells are blocked. They always know which (unblocked) cells they are in, sense the blockage status of their four neighboring cells, and can then move to any one of the unblocked neighboring cells. Their task is to move to a given goal cell. Our game characters find (with Adaptive A*) and then follow a cost-minimal presumed unblocked path from their current cell to the given goal cell, where a presumed unblocked path is one that does not pass through cells that are known to be blocked. (Note that they can search forward from their current cell to the given goal cell or backward from the given goal cell to their current cell.) Whenever the game characters observe additional blocked cells during execution, they add them to their map. If such cells block their current path, they find and follow another cost-minimal presumed unblocked path from their current cell to the given goal cell, and repeat the process until they either reach the given goal cell or all paths to it are blocked. Figure 3 shows the resulting A* searches and Figure 4 shows the resulting Adaptive A* searches for a simple navigation example. The black circle is the game character. Black cells have been observed as blocked. The arrows show the planned paths from the current cell of the game character to its goal cell, which is in the lower right corner. All search methods break ties between cells with the same f-values in favor of cells with larger g-values and remaining ties in the following order, from highest to lowest priority: right, down, left and up. All cells have their h-value in the lower left corner. Generated cells also have their g-value in the upper left corner and their f-value in the upper right corner. Expanded cells are shown in grey. For Adaptive A*, expanded cells have their updated h-values in the lower right corner. Note that A* re-expands the three cells in the bottom row because the h-values are misleading. Adaptive A* avoids these re-expansions since it updates the h-values.

Figure 5: Excerpt: Description of Adaptive A* for the Search Project (Short Version)

contains such an example. One comment on ratemyprofessors.com after fielding the project in the graduate introduction to artificial intelligence was "Beware the "warm up project." Its [sic!] like warming up by running a marathon, but the class is interesting!" We improved the project description and fielded it again in the graduate introduction to artificial intelligence, where is was evaluated with a detailed questionnaire that included questions such as "was the project easier, equally difficult or harder than 2-week projects in other computer science classes at USC" (7:8:8), "was the project less time-consuming, equally time-consuming or more time-consuming than 2-week projects in other computer science classes at USC" (4:8:10), "did the project achieve its objective of allowing you to understand A* better" (23:0:2) and "should we offer the project again" (24:2:1). Some students suggested that we provide a code skeleton and de-emphasize the proofs.

Machine Learning

We are developing a similar step-by-step project for machine learning, namely one that uses genetic algorithms to evolve game characters that survive in a hostile environment, inspired by (Laramee 2002), and that use neural networks for the recognition of static hand poses and gestures, inspired by (Mitchell 1997). However, we are also experimenting with

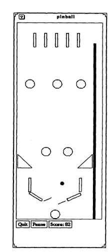

Figure 7: Pinball Simulator (Winstead & Christiansen 1994)

a more open-ended project for machine learning where the students use machine-learning methods to control the simulation of a pinball machine, see Figure 6. Playing pinball provides an interesting control problem (namely, how to best affect the ball) that combines high-level planning with low-level control in a time-constrained and not completely predictable environment. Insight into this control problem can potentially be used to solve more complex control problems, such as flying helicopters. It appears to be relatively simple: First, we use a simple pinball machine that does not require state information other than the state of the ball, which is given by four floating point numbers that

Write a learning system that learns to play pinball well on a simple pinball simulator that we will provide you with. First modify the layout of the pinball table so that the ball gets easily lost when it ends up in the wrong part of the table. Then, rearrange the bumpers so that the learning system can score high in that part of the table. The task of the learning system is to get as high a score as possible with, say, 50 balls. It can observe the coordinates of the ball, but has no prior knowledge of the layout of the pinball table. Now the learning system has to trade-off between shooting the ball where it can learn something new, shooting the ball where it can score high, and shooting the ball where it is safe. How does your learning system handle this trade-off? For example, does it reason about the trade-off explicitly or implicitly? You can look up prior work on this problem, including (Johnson 1993; Winstead & Christiansen 1994; Winstead 1996), but be creative and come up with ideas on your own (either something completely different or something that extends the previous ideas).

Figure 6: Excerpt: Pinball Project

describe its position and velocity. Second, there are only two binary user inputs that describe which flipper buttons have been pressed. However, the control problem is challenging to solve. It is therefore not surprising that we know of three undergraduate projects (often honors projects) that attempted to learn when to flip the flippers to either keep the ball in play for as long as possible or to score as high as possible, namely a student at Massachusetts Institute of Technology who used both stochastic hillclimbing and Hoeffding races (Johnson 1993), a student at Tulane University who used reinforcement learning (Winstead & Christiansen 1994; Winstead 1996), and our own student who used neural networks. We use a cleaned-up version of the simple pinball simulation of Nathaniel Winstead, see Figure 7. We added two kinds of interfaces, a user interface that allows people to play the simulation and a software interface that allows one to write controllers. We provide a simple controller that repeatedly flips both flippers up and down when the ball crosses an imaginary line from above that is close to the flippers, for a total of less than 1500 lines of C++ code. People can play much better than this simple controller (even if they have never played the pinball simulation before), yet it is challenging to write better controllers. We have used this project as an undergraduate honors project, as an introductory graduate research project and as the last project in graduate artificial intelligence classes that covered a variety of machine learning and search methods, including decision trees, neural networks, reinforcement learning, probabilistic search with Markov decision processes, and hill-climbing algorithms (such as simulated annealing and genetic algorithms). We then left it up to the students to attack this control problem with any method that they learned in the class. We ran early trials of this project (where the students came up with very creative solutions) but yet have to use it in artificial intelligence classes at USC.

Conclusion

We reported on the efforts of USC to teach computer science and artificial intelligence with games. We are having fun in the process! The future will show whether we manage to achieve our objectives, namely to increase retention and educate better computer scientists. We will soon put projects and additional material onto our webpages at "**idm-lab.org/gameai**".

References

Buro, M. 2003. Real-time strategy games: A new AI research challenge. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 1534–1535.

Business Roundtable. 2005. Tapping America's potential: The education for innovation initiative.

Cliburn, D. 2006. The effectiveness of games as assignments in an introductory programming course. In *Proceedings of the Annual ASEE/IEEE Frontiers in Education Conference*, 6–10.

Fullerton, T. 2006. Play-centric games education. *IEEE Computer* 39(6):36–42.

Hearn, R. 2004. *Tribute to a Mathemagician*. A K Peters. chapter The Complexity of Sliding Block Puzzles and Plank Puzzles.

Holte, R.; Mkadmi, T.; Zimmer, R.; and MacDonald, A. 1996. Speeding up problem solving by abstraction: A graph oriented approach. *Artificial Intelligence* 85(1-2):321–361.

Hordern, E. 1986. *Sliding Piece Puzzles*. Oxford University Press.

Johnson, M. 1993. Algorithms for pinball simulation, ball tracking, and learning flipper control. Technical report, Massachussets Institute of Technology. Bachelor's Thesis.

Koenig, S., and Likhachev, M. 2005a. Adaptive A* [poster abstract]. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems*, 1311–1312.

Koenig, S., and Likhachev, M. 2005b. Fast replanning for navigation in unknown terrain. *Transactions on Robotics* 21(3):354–363.

Laramee, F. 2002. *AI Game Programming Wisdom*. Charles River Media. chapter Evolving the Perfect Troll.

Mitchell, T. 1997. *Machine Learning*. McGraw-Hill.

Winstead, N., and Christiansen, A. 1994. Pinball: Planning and learning in a dynamic real-time environment. In *Proceedings of the AAAI-94 Fall Symposiumon Control of the Physical World by Intelligent Agents*, 153–157.

Winstead, N. 1996. Some explorations in reinforcement learning techniques applied to the problem of learning to play pinball. In *Proceedings of the AAAI-03 Workshop on Entertainment and AI / A-Life*, 1–5.

Zyda, M.; Mayberry, A.; McCree, J.; and Davis, M. 2005. From viz-sim to vr to games: How we built a hit game-based simulation. In Rouse, W., and Boff, K., eds., *Organizational Simulation: From Modeling and Simulation to Games and Entertainment*. Wiley Press. 553–590.

Zyda, M.; Lacour, V.; and Swain, C. 2008. Operating a computer science game degree program. In *Proceedings of the Game Development in Computer Science Education Conference*.

Zyda, M. 2006. Educating the next generation of game developers. *IEEE Computer* 39(6):30–34.

Zyda, M. 2007. Creating a science of games. *Communications of the ACM* 50(7):26–29.