William I. Grosky
University of Michigan-Dearborn

Games in the Classroom: Using Games as a Motivator for Studying Computing: Part 1

Andrew M. Phelps, Christopher A. Egert, and Jessica D. Bayliss
Rochester Institute of Technology

This article, the first in a two-part series that explores using games as a gateway to studying computing in the classroom, explores the need for a motivator in today's educational environment and places games in the context of constructivist learning approaches. In addition, it provides an overview of several recent approaches and looks at issues associated with student perception, exploring how existing paradigms relate to these perceptions.

The second article will focus not only on the underlying educational models for which games are particularly well suited, but also on the results of our approach. In addition, the second article will address gender issues as they relate to gaming in the classroom. In addition to this two-part article, we will present our own experiences related to the topic in the Multimedia at Work column in the next issue.

Editor's Note

It's no secret that undergraduate computer science enrollment, which has suffered through one of its periodic downturns, seems to have bottomed out but is now on an upswing. This cyclic behavior has been occurring for many years now, producing many exciting ideas concerning how to revamp introductory computer science courses to make them more exciting and relevant, and to show beginning students that computer science entails more than just programming. Georgia Tech, one of the active participants in this revamp, has developed the concept of *threads* (a means to connect chunks of related knowledge across different courses) and is devising techniques to enrich beginning courses using minirobots and multimedia. The present article, written by Andrew Phelps and his group at the Rochester Institute of Technology, describes a parallel effort to use gaming as a way to improve learning and to demonstrate to students that computer science is indeed exciting and cool.

—William I. Grosky

Overcoming the crisis

For the past several years, computer science has faced a crisis revolving around the perception and efficacy of the field, with computing departments around the US reporting troubling statistics regarding recent trends. For example, the 2006–2007 Taulbee Survey reported that in the Fall of 2007, there were approximately 50 percent fewer entering students for computer science programs in comparison to the Fall of 2000.[1] Some of this downturn is a result of perceptions related to the job industry, as potential students are still influenced by the industry problems from the early part of this decade, while others are concerned with current economic conditions and continued worries regarding downsizing and outsourcing.

In addition, potential students are prejudiced by the perceptions of people in computing, still associating computing with antisocial, nerdy behavior in which practitioners sit behind a desk all day, isolated from others, and work on uninteresting and irrelevant problems that provide little personal or societal gratification. Even when students are attracted to computing, it's often difficult to keep their interest, particularly through introductory coursework that imparts needed skills before more interesting problems can be explored.

Frequently, a student's first exposure to computing is through introductory programming, which requires the student to learn syntax, design, logic, debugging, and more, without adequate instruction in the relationship of this introductory material to long-term objectives. Students might not see the relevance of introductory experiences, as many introductory examples are not synonymous with the student's expectations, and thus students cannot make connections between the exercises—such as

1070-986X/09/$25.00 © 2009 IEEE

Published by the IEEE Computer Society

text-based problems, business formulas, and text manipulation—and what they have already experienced as the richness of computing.

It's no wonder, then, that failure rates in computing courses over the first year are staggeringly high, indicating that students are frustrated, disenfranchised, and disconnected from their intended career path. We believe that one way to counteract these failure rates is to engage student interest by associating computing with a domain that speaks to the student's generation. One such domain is that of video game design and development.

The influence of video games on today's youth represents a larger trend of technology that inspires new forms of communication, social interactions, information retrieval, and entertainment. Video games are not just a short-term phenomenon. They are iconic, and permeate literature, art, and cinema. Video games also represent an outlet of creative expression even by those who do not produce games themselves, inspiring fan fiction, art, and other experiences. These observations point to how students perceive technology as a creative medium through which interactive experiences, stories, and exploration can be experienced on a personal and social level.

It's no wonder, then, that when a professor speaks about technology careers, students appear to be captivated by the idea of making the next triple-A game title. Clearly, using games in the classroom can be a wonderful way to illustrate how computer science can integrate with and augment creative expression. After all, game development is not undertaken in a technology vacuum. Practitioners are expected to interact with artists, musicians, writers and many others whose professions are aligned with the creative arts and content production. In addition, game development provides an appropriate context for computing concepts, such as program design, data structures, graphics programming, artificial intelligence, language theory, human–computer interaction, computer music, optimization, multicore and parallel processing, algorithms, and the theory of computation.

However, despite the potential video games have to offer, there are several barriers that must be overcome to use game technology in the classroom. Effectively using video games in the classroom means bridging the generational gap between faculty and students, dismissing faculty perceptions that games are not suitable for education, providing adequate technology for what students consider to be real game development, and eliminating negative stereotypes associated with commercial video game development.

Games as a motivator

To understand a student's needs in approaching computer science, it's helpful to examine the role of cognition and learning. One approach to cognition is to classify the different behaviors associated with learning, and to classify the various tasks and objectives in the introductory classroom. Bloom's taxonomy is frequently used as a starting point for such endeavors, and can help educators analyze instructional design and assess class activities.[2] Classifications built upon such taxonomies rely on a pyramid structure, with knowledge at the bottom, followed by understanding, application, analysis, synthesis in the middle, and evaluation at the top. The tenet is that items toward the bottom involve rote memorization whereas those toward the top involve deeper levels of understanding.

Classification schemes can form the basis for cognitive models that attempt to define learning processes. Corresponding to Bloom's lower levels, cognitive models can help educators understand how students internalize core concepts, such as variable assignment, selection, looping, arrays, and other language constructs. They can also help educators understand concepts such as problem comprehension, task decomposition, and the means by which introductory programmers implement and evaluate the effectiveness of their solutions.

Games are an intuitive and natural route from the bottom to the top of Bloom's taxonomy. While most introductory courses focus on the lower three portions of the taxonomy (knowledge, understanding, and application), the use of games in an introductory programming course leads naturally to synthesis. Students find that after they have constructed a game, they can use their computing knowledge to critique the games they already play. Doing so allows students to create links between classroom activities and activities performed outside the class.

Classification by affective domain

Another approach to understanding student learning is to classify learning on the basis of the affective domain, as characterized by the

second volume of Bloom's taxonomy.[3] The affective domain model describes changes in student interest, attitudes, behavior, and values, with key features that include how students are sensitized to stimuli and phenomena, how students become motivated to participate, how students ascribe value to behavior, how students organize values and resolve conflicts, and how students internalize values. Games are a natural match to all levels of the affective domain.

Introductory courses go to great lengths to have students understand cause and effect during design and development. The highly visual and interactive nature of games and their relative complexity can help in this understanding by creating appropriate scenarios that act as an anchor for discussion and a way to for students to internalize key concepts. Games provide a familiar anchor for exploring the affective domain, as students tend to approach problems with preconceived notions about games. Instructors can leverage or challenge such knowledge, depending on the particular programming activity at hand.

Constructivist learning model

Another approach to the introductory classroom is to change the interaction model by decentralizing the authority of the instructor.[4] This model, based upon several core ideas that relate to power in the classroom, positions the learner's knowledge and experiences as a starting point for new learning. As students approach a particular problem, individual learning and knowledge are reflected and refined through classroom participation. The constructivist model relies on tasks that are more complex than simple problems. For the model to be effective, the tasks must consist of real-world scenarios in which discovery and action can have compelling effects.

Those who use the constructivist model in the classroom seek to cultivate knowledge and concepts that are uniquely identifiable and personal to the learner. Some educators have suggested techniques to integrate constructivist principles directly into the introductory programming curriculum, while others have suggested general guidelines to maximize the impact of constructivist theories. Investigations into the impact of constructivist techniques in the classroom have revealed a range of outcomes, from anecdotal evidence of classroom success to empirical studies showing positive results.

Some educators find constructivist techniques challenging because they displace a central authority figure who can maintain control of classroom interaction and a specific approach to the materials. As such, the constructivist classroom requires a great deal of planning and implementation of effective mechanisms to provide nonintrusive guidance to learners. Despite these challenges, the constructivist classroom appears to be well suited to learning about game development, especially because, as a general rule, game developers must trade between design and optimization considerations, a factor that leads to more than one solution for a given problem.

The constructivist classroom allows the student to shift his or her knowledge from game consumer to game developer, transforming the ideals of play strategy, game choices, and outcomes into the logic that produces a game solution. In this way, games can aid in discussions about choices and intended outcomes. In addition, games provide an appropriate level of complexity for constructivist exploration. Even simple arcade games from the 1980s provide enough richness and challenge when dissected into their basic principles.

Self-efficacy in the classroom

Another approach to the study of introductory computing is to focus on self-efficacy.[5] The concept of self-efficacy entails beliefs regarding one's own abilities. Self-efficacy is a function of multiple inputs, including current skills and proficiencies; physiological states; psychological well-being; the ability to understand and internalize observed phenomena; and the influence of other people, such as instructors and peers. Thus, self-efficacy can vary on the basis of the type of student activities as well as the mode of student participation.

Games might have a positive impact upon self-efficacy because participants can interact with games on several levels. As students construct games, they receive immediate feedback framed within a functional framework they can easily understand. That is, the concrete nature of game construction can serve to minimize feelings related to lack of self-confidence because scenarios that occur when mistakes are made can be contextualized for the learner.

In addition, the concept of self-efficacy can be used itself in conjunction with games to expose learners to areas that are less comfortable for them or about which learners feels they

have less knowledge. Students can adjust their comfort levels by doing tasks in which they feel higher levels of self-confidence instead of those that create anxiety and doubt. Even within novice classrooms, instructors can rely on tasks related to higher self-efficacy in terms of game design, interactive choices, and game outcomes to help learners achieve their goals.

Programming paradigms

Another way of looking at introductory programming is in terms of the paradigm used. Over the past decade, instruction has shifted from the imperative to the object-oriented paradigm. Although the term *objects-first* is not universally known, the ACM/IEEE Joint Task Force for Computing Curricula provides a working model for the concept.[6] The objects-first approach to object-oriented programming begins with concepts of objects and inheritance, followed by traditional control structures, such as selection and loops. Students learn traditional programming concepts within the context of objects, with both design and programming sharing an equal role. Others have gone further by defining the objects-first approach as meaning the exploration of encapsulation, inheritance, and polymorphism in the introductory course.[7]

Despite the popularity of object-oriented languages, there is much debate about when object-oriented principles should be introduced.[8] Some practitioners believe that object-oriented concepts should be taught after traditional imperative concepts to reduce the cognitive demands upon the student. Others suggest introducing objects early in the process or state that the objects-first approach should be taught in a manner in which task decomposition, responsibility management, delegation, and programming share equal classroom time.

We believe games can be effective in helping students learn several paradigms and can be particularly useful with an objects-first approach. Games can help students learn object-oriented principles and abstractions, such as objects that manage game elements, objects that act as containers for groups of game components, objects that act as event handlers for input or timed operations, and objects that maintain state and perform appropriate transitions within games. Moreover, game construction can help teach the concepts of inheritance and polymorphism, which scale well to the complexity of game design. For example, game components controlled by both the player and the computer can use inheritance to derive new functionality in terms of motion or other behavior, while effective use of polymorphism can help maintain the collection of game elements uniformly.

Games in the classroom

As is the case with approaches to teaching and learning in general, there are several ways instructors can use video games as motivators in the classroom. At a recent conference on game development and education, participants explored several approaches.[9] One technique is to provide sufficient support through libraries, examples, and applications to allow students to work with a particular concept. For example, some universities use predeveloped games to teach students introductory concepts, such as loops and arrays. By using an existing game framework and sets of libraries, students can experiment by changing the game application and library calls through a process of trial and error, creating deeper understanding of the subject matter.

Another approach is to use commonly played, well-understood games, such as tic-tac-toe, hangman, and other favorites, as a means of understanding state, abstraction, and event handling. Some institutions use commercial and open-source frameworks, such as Adobe Flash, Adobe Director, Microsoft XNA, and YoYo Games GameMaker as playgrounds for teaching introductory programming concepts in interactive environments. Some approaches even use simple 2D game projects, such as Asteroids and Tetris, to illustrate introductory programming examples.

Another technique is to use game concepts either in a second-semester or second-year course as a way to ground critical concepts learned in prior work. This approach can be implemented in conjunction with game experiences from the first year, or as a standalone follow-up to an introductory course. For example, some programs use the downstream courses as a way to ground introductory material while teaching new material in areas such as interface control, artificial intelligence, animation, collision detection, mechanics, and finite state machines. Some teachers use strategy and puzzle games, such as Scrabble, Othello, and Connect Four, in the second semester as a way to help students learn data structures such as stacks, queues, and trees.

Still another approach is to have students program in a rich, interactive environment

that is similar to what would be experienced in commercial game development, with appropriate attention given to level of detail, interaction clues, and task contextualization. An excellent example of one such environment is Karel the Robot, a project designed to allow students to build functions to control movements and add complexity by reusing behaviors.[10] The Karel project gives students the ability not only to design relatively complex behaviors in a short period, but also to make changes to the robot's programming to observe the results.

Another rich, interactive environment is the Alice project, which is designed to allow students to explore introductory programming.[11] In the Alice project, students create Java objects using a visual interface, selecting language constructs from graphical menus that help minimize syntactical errors. Students don't have to understand 3D mathematical principles to program graphically rich applications in the Alice system. Instead, students can concentrate on the thinking skills that are critical for programming and logic implementation.

In a similar manner, the Rapunsel project provides introductory programmers with a microworld that gradually introduces them to computing principles.[12] Rapunsel is designed so that students can make mistakes. Instead of causing critical errors, however, these mistakes can be leveraged into valuable learning experiences that help foster student empowerment and self-management. The Rapunsel project specifically addresses gender differences in programming education and tailors experiences to help engage female students.

Conclusion

We believe that for today's student considering a career in computing, video game development can provide a compelling learning framework that helps refine computer science skills through tangible experiences. Video games provide concrete, real-world examples of computing that extend beyond operating systems and other such intangibles, intersecting with both the academic field and the general public. When properly handled by instructors, video games can give entering students a solid framework for relating their studies socially to their peers and for grounding their intellectual curiosity in the functional rather than the abstract. In our next article, which will appear in the July–September issue of MultiMedia, we plan to explore the educational models associated with these observations and provide an assessment of the effectiveness of our approach to a games-based curricula. **MM**

References

1. S. Zweben, ''2006-2007 Taulbee Survey: Record Ph.D. Production Exceeds 1,700; Undergraduate Enrollment Trends Still Unclear,'' *Computing Research News,* vol. 20, no. 3, 2008.
2. B. Bloom et al., *Taxonomy of Educational Objectives: The Classification of Educational Goals: Cognitive Domain,* Longman, 1956.
3. D. Krathwohl et al., *Taxonomy of Educational Objectives: The Classification of Educational Goals: Affective Domain,* McKay, 1964.
4. J.G. Brooks and M.G. Brooks, *In Search of Understanding: The Case for Constructivist Classrooms,* Assoc. for Supervision and Curriculum Development, 1999, p. 136.
5. V. Ramalingam et al., ''Self-Efficacy and Mental Models in Learning to Program,'' *Proc. 9th Ann. Conf. Innovation and Technology in Computer Science Education,* ACM Press, 2004, pp. 171-175.
6. Joint Task Force on Computing Curricula, *Computing Curricula Final Report: Computer Science,* 2001; http://www.computer.org/portal/cms_docs_ieeecs/ieeecs/education/cc2001/cc2001.pdf.
7. P. Ventura, *On the Origins of Programmers: Identifying Predictors of Success for an Objects-First CS1,* doctoral Dissertation, Computer Science and Engineering, University at Buffalo, SUNY, 2004.
8. K.B. Bruce, ''Controversy on How to Teach CS 1: A Discussion on the SIGCSE-Members Mailing List,'' *SIGCSE Bulletin,* vol. 36, no. 4, 2004, pp. 29-34.
9. *Proc. 3rd Int'l Conf. Game Development in Computer Science Education,* ACM Press, 2008.
10. R. Pattis, *Karel the Robot: A Gentle Introduction to the Art of Programming with Pascal,* John Wiley and Sons, 1981.
11. S. Cooper et al., ''Teaching Objects-First in Introductory Computer Science,'' *Proc. 34th SIGCSE Technical Symp. Computer Science Education,* ACM Press, 2003, pp. 191-195.
12. M. Flanagan et al., ''Values at Play: Design Tradeoffs in Socially-Oriented Game Design,'' *Proc. SIGCHI Conf. Human Factors in Computing Systems,* ACM Press, 2005, pp. 751-760.

Contact author Andrew M. Phelps at andy@mail.rit.edu.

Contact editor William I. Grosky at wgrosky@umich.edu.