Qibin Sun
Hewlett-Packard

Games in the Classroom at the Rochester Institute of Technology: A Case Study

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

We have worked together to create curricula, microworld applications, and game examples designed to motivate introductory students in our formal game design and development programs and in our traditional computing programs, such as computer science and information technology. This article examines three distinct approaches that we have incorporated in our coursework: an introductory programming sequence based on game examples, the incorporation of game-based projects into a required core course in media-centric programming, and the creation of a microworld environment that mimics the look and feel of online game environments.

First year: reality and programming together

As educators, the first-year introductory programming sequence provides our first opportunity for introducing students to their chosen computing discipline. The challenge of engaging the introductory student is multifaceted: at one level, students must learn the syntax of programming; at another level, they must learn to leverage the particular language paradigm in which they are working. In addition, issues of abstraction, algorithmic thinking, software construction, and debugging strategies dominate the introductory classroom. What sometimes gets lost in the process is the link between the programming process and the usefulness of programming as an expressive tool for a particular problem domain. To address this problem, we have examined domain-centric approaches to introductory programming, specifically for students wishing to study game design and development as a discipline.

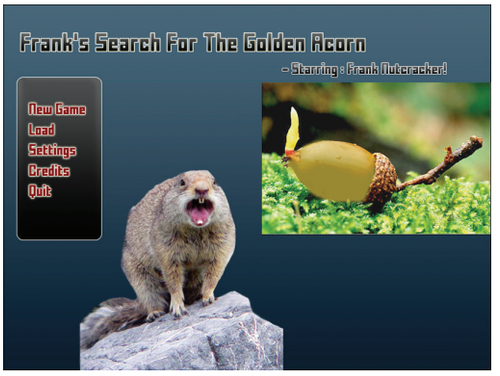
As an alternative to a more traditional sequence in computer science, we offer the Reality and Programming Together (RAPT) sequence of courses to first-year students in the bachelors of science in game design and development program[1,2] at Rochester Institute of Technology. RAPT presents students with the same learning outcomes as the more traditional first-year coursework, but within the context of game design and development. From the first day of the course and through the entire year, students learn key introductory computing concepts for game programming. The domain of computer game design and development not only links students to their chosen discipline, but also demonstrates concepts in an environment with which the students are intimately familiar.

The first course is a 10-week experience in which students learn fundamental skills, such as programming language syntax; semantics; object-oriented design and implementation; software lifecycle; and control structures, such as selection and loops. These topics are presented while the class endeavors to move from critical consumers of games and entertainment technologies to informed producers of interactive, compelling experiences.

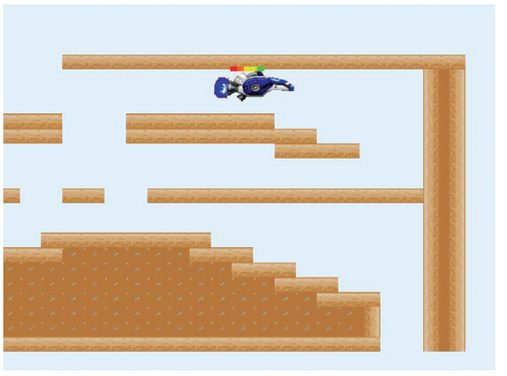
The first week of the course starts with an introduction to problem solving as well as an introduction to programming tools, including integrated editing environments, interpreters, and compilers. In the first assignment, we ask students to compile an existing game program, with emphasis on the use of the tools and on the exploration of cause-and-effect relationships when altering elements of a preexisting game. As the course progresses, students learn about key concepts, such as variable assignment, data types, and expressions, while they also learn about object-oriented structure and design. Object-oriented structures include

1070-986X/09/$26.00 © 2009 IEEE

Published by the IEEE Computer Society



(a)



(b)

Figure 1. (a) Sample game from the third RAPT course. (b) This game, **Red Ship Blue Ship**, *was later polished and submitted to the Independent Games Festival.*

examination of critical concepts, such as differentiation of classes and objects, methods, properties, and class relations. Also in the first week of the course, students begin to learn concepts such as object references, parameters, return values, selection, logic operations, and much more.

In the first few weeks of the course, students create projects that require they not only understand APIs and game frameworks, but also make creative decisions on their own that directly impact the experiences they are producing. During this period, students work with games that are based on numerical manipulation, decision making, and interaction. In the middle of the course, students learn about iteration, accessors, mutators, encapsulation, overloading, and scope. The course culminates with an exploration of inheritance. After a mere 10 weeks, students have an appreciation of programming and are able to build personally meaningful applications that can be shown to family and friends.

Along with exploring the fundamentals of programming, the first RAPT course is designed to address students' interests in game development as a profession. The course introduces students to fundamental software engineering techniques used in the game industry, including standards of design, documentation, and testing. The first RAPT course also provides students with opportunities to critique games, including commercial games as well as those created by their classmates. Students make connections between the two and draw analogies between commercial techniques and their relationships to introductory tasks. Incorporation of games into the course in this fashion also provides opportunities to discuss optimization and debugging strategies early in the educational process.

The second and third courses in the RAPT sequence are designed to enhance student understanding through teaching more advanced concepts. The second course starts with an examination of arrays versus collections, generics, and file I/O. The course advances into simple data structures, such as linked lists, stacks, and queues. The second course addresses threading, event processing, and GUI programming. Throughout the course, we take a just-in-time approach to the introduction of design patterns and architectures for game systems.

The third course, which is devoted mostly to algorithms and data structures necessary for game design and development, explores complex systems and program efficiency. The third course emphasizes teamwork as an important part of the game-development process, and culminates in a group programming experience in which students have well-defined roles and responsibilities. Just as in the first course, students heavily invest in their game creations and often display their work to family and friends. The team experience not only helps students create appropriate connections with peers, but also serves as positive reinforcement as students motivate each other to extend and expand their introductory work outside of the classroom. One student team even submitted its game to the International Games Festival contest. Figure 1 shows sample games.

Interactive digital media

An example of the use of games after the introductory sequence is a course titled Interactive Digital Media. We designed the course to teach students how to programmatically interact with and control specific media types,

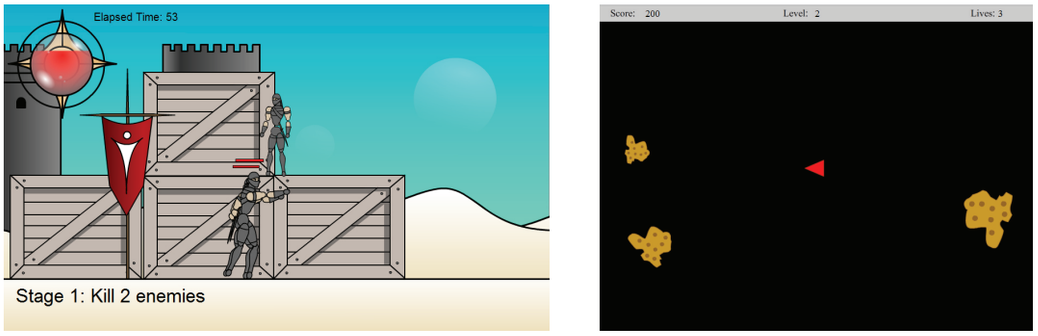


Figure 2. Sample games created in a downstream Flash-based course entitled Interactive Digital Media.

including text, vector animations, components, images, sounds, and video. One of the major assignments in the course is the construction of a classic 2D video game. The section on games begins with a brief history of classic games from the 1970s to the early 1990s. In the section, students are exposed to games they might have only experienced through pictures and Web sites. In addition, students explore design considerations of classic games to motivate discussions about design techniques and possible pitfalls in the students' own implementations.

Following this introduction, students take part in a decomposition exercise for a traditional 2D arcade game. In the past, we've employed games such as carnival shooters and the ever-classic *Asteroids*. Students learn to identify key game functionality related to graphical assets, such as the player's representation in the game, the computer-controlled items that act as the opponent, the user interfaces, and the graphics related to game state. Students also learn to addresses issues pertaining to roles and responsibilities for nonvisible items, such as content managers, control logic, and preloaders.

Following this instruction, faculty members then engage students in a guided exercise focused on the construction of a complete game from scratch. The exploration of implementation issues starts with the development of graphical assets and an exploration of their relation to programmatic objects designed for the game. The instructors then teach about the role of external data, most commonly implemented through an XML schema, to tune the overall game experience without altering and recompiling the game.

Fundamental concepts, such as keyboard control classes, collision management, asset management, time management, state maintenance, and state transition, are taught through game construction. Considerable discussion surrounds the choices made for encapsulation, appropriateness of class and object responsibility, delegation of task from asset classes to managers, and event management. Other topics of discussion include optimizations for collision detection and overall game performance as well as debugging strategies when errors occur. Finally, the instruction includes discussion of soundtracks, sound effects, and video as part of a game experience.

We designed the project to be open-ended. Students are required to either propose their own game or base the design on an arcade classic. Furthermore, students are required to identify potential problems in the design and implementation of their game and must discuss viable options before proceeding to development. The assignment structure strictly enforces several checkpoints at which students must communicate with their instructor and peers, gather feedback on both the technology and game play of their project, and discuss their decisions to either incorporate or dismiss suggestions. The instructor encourages students to discuss their problems and possible solutions in and outside the classroom.

Students develop the game project in the Adobe Flash environment and use the ActionScript 3.0 programming language and Flash Player classes for the implementation. Figure 2 shows samples of completed work.

Multi-User Programming Pedagogy for Enhancing Traditional Study

We have examined the use of a microworld-like environment for the exploration of introductory programming concepts. The Multi-User Programming Pedagogy for Enhancing Traditional Study (M.u.p.p.e.t.s.) is a collaborative virtual environment specifically designed to support

(a)

(b)

Figure 3. (a) The Multi-User Programming Pedagogy for Enhancing Traditional Study integrated development environment, and (b) an IDE with an instantiated object.

the needs of introductory programmers.[3-10] Our goal with the M.u.p.p.e.t.s. system is to augment traditional approaches in the classroom while simultaneously providing students and faculty with a motivating environment to explore critical programming concepts.

At a technical level, M.u.p.p.e.t.s. provides rich interactive graphics, integrated development and debugging tools, and basic collaborative mechanisms for sharing objects and code among learners. M.u.p.p.e.t.s. allows students to explore both objects-first and object-early programming by allowing for early investigation of encapsulation, inheritance, and polymorphism. Through M.u.p.p.e.t.s., students have the ability to construct classes and instantiate objects within a private workspace. Here, students can make appropriate adjustments, refining the design and implementation of their virtual assets.

When a student is satisfied with his or her creation, the resulting materials can be introduced to the public. Users of M.u.p.p.e.t.s. technology can interact with each other in the public space within the virtual world, and can share code and fully realized objects. From an educator's standpoint, the technologies afforded by the M.u.p.p.e.t.s. system have the potential of being mapped to constructivist and cooperative learning opportunities in and outside the traditional classroom.

The M.u.p.p.e.t.s. system is unique in that it's specifically designed to progress into downstream courses with the learner. The M.u.p.p.e.t.s. system supports Java and C# in their entire form, without modification. By supporting languages in their entirety, the student has access to all constructs that he or she might encounter in the programming world. As such, any program written in Java or C# can be modified to execute in the M.u.p.p.e.t.s. environment (and will in fact operate in unmodified fashion, but will simply produce no graphical output within the virtual environment beyond the console).

The M.u.p.p.e.t.s. system provides development tools to the user from within the interactive environment. As such, the user invokes the integrated development environment (IDE) while simultaneously interacting with the world. The editor allows the student to define new classes as well as issue interactive commands to the world. The IDE offers the student several features, such as color coding, code completion, and contextual help, that are normally present only in high-quality commercial and open source software. The M.u.p.p.e.t.s. system also provides students with access to an integrated debugger. Preliminary testing of the system indicates that this dual-interaction between the IDE and the ever-present world is a key attractor to student programmers: their perception is that by not having to leave the environment to recompile their creations, the process is somehow more inclusive of their designs. An example of the M.u.p.p.e.t.s. environment is shown in Figure 3.

M.u.p.p.e.t.s. incorporates a 3D graphics framework that is based upon practices and techniques developed in the gaming industry. The system provides support for such features as modern shading languages, stencil and depth shadows, and normal mapping. Students can render their virtual world in OpenGL and DirectX, and as a result can learn about differences in graphics systems. M.u.p.p.e.t.s. allows imports from professional content-creation tools, such as Autodesk's Maya and 3D-Studio Max, and allows students to experiment with

techniques such as terrain generation, particle systems, lighting, advanced texturing, and other current techniques.

In the M.u.p.p.e.t.s. system, the collaborative features supported through a multiuser server are modeled from current practices in massively multiplayer environments. M.u.p.p.e.t.s. provides different levels of granularity for access and control of the shared system, allowing for different levels of synchronization between networked objects and environments. M.u.p.p.e.t.s. also provides direct support for several critical tenets of the objects-first approach. The system is highly dependent on the concepts of encapsulation, inheritance, and polymorphism. For example, when operating within the M.u.p.p.e.t.s. system, students are immediately exposed to inheritance by extending the generic `MuppetsObject` class. The M.u.p.p.e.t.s. environment provides immediate feedback for the exploration of class relationships, such as composition, association, and dependency. In addition, the environment can provide a testbed for exploring the use of design patterns in the construction of software.

M.u.p.p.e.t.s. also supports iterative refinement in the development of classes. Students can immediately introduce a new object into the M.u.p.p.e.t.s. world for private testing once the class has been built within the IDE. From within the environment, the student instantiates the object and is presented with immediate feedback. If the construction of the object cannot be accomplished, the system presents the student with an error message in the 2D console and creates a 3D placeholder for the object in the virtual world. The programmer can continue to interact with objects in-world as well as from the IDE. If the student is unsatisfied with a class for any reason, he or she can redefine the object. Students can overwrite or publish user objects created in this way at each stage of refinement.

We have already explored using M.u.p.p.e.t.s. to address key concepts in an objects-first curriculum driven by constructivist learning models. In addition, we've experimented with laboratory experiences that explore issues related to composition and association. For example, students learn to differentiate composition (complex objects created via construction of simple parts) and association (references to other objects stored for later use during the construction of the initial object). Instructors have students explore the concepts of composition and association by asking them to select single items from the real world and describe how they can be created from fundamental parts. The instructor can use the opportunity to reinforce prior lessons in identifying properties and methods for objects, and students can then begin the construction of a well-understood object, such as a simple 3D face, exploring the notion that even simple items, such as eyes, noses, and mouths, can be complex when determining what gets created first and how each piece is related to the other.

M.u.p.p.e.t.s. can provide opportunities for students to work in teams to create the fundamental parts for characters in an iterative design cycle in which students redesign ideas and deliverables. The characters can then be used in later exercises that explore common design patterns and in lessons about the mathematics and believability of motion. At the end of a relatively short period, a team of students can produce a rich corpus of characters and interactive virtual objects for later use, while having a profound and educational experience that underscores the basic tenets of association and composition.

As another example of the use of games in the classroom, students were provided with code and several programming interfaces to help implement the artificial-intelligence system for a tank.[4] Students had to design algorithms to control features such as moving the vehicle, turning the vehicle, and firing the weapons system. Students had to consider various behaviors, such as collision detection and avoidance, aggressive behaviors, and defensive behaviors. The activity of designing and developing the tank artificial intelligence in M.u.p.p.e.t.s. was both cooperative and competitive: students had to work together in teams to develop the system and competed with each other to see whose design was the most effective. Analysis of student experiences showed that students felt they learned more using the M.u.p.p.e.t.s. environment in this assignment than they did using other programming interfaces in other similar assignments.[4]

Conclusion

Although we have employed games as a motivational technique in several courses, the assessment of games on self-efficacy, course outcomes, concept transference, and program

retention is still in the preliminary phases. For two years, we collected and analyzed data from the RAPT programming sequence.[1] During RAPT's deployment, we found that student grade distributions were significantly different from regular section grade distributions in computer science 2 and 3 courses ($p < 0.005$, chi-squared, two tailed). Significance dropped a bit between the groups when students exited the RAPT sequence and took computer science 4 ($p < 0.02$, chi-squared, two tailed).

Fifty-eight percent of the RAPT students received a grade of A from computer science 4 followed by 28 percent with Bs. For regular sections, 25 percent received As and 29 percent received Bs. Thirty-six RAPT students and 51 regular students took computer science 4. Retention in the computer science major improved in RAPT with an average retention rate of 93 percent ($n = 44$) when compared to the regular section rate of 57 percent ($n = 164$). Such preliminary reports are encouraging and also demonstrate that we have a long way to go in understanding the motivational potential of games.

Since the establishment of the bachelors of science in game design and development, the Rochester Institute of Technology (RIT) has seen unprecedented interest in the program, which speaks directly to the overall interest in computing, particularly when computing is defined in less traditional terms. Despite a great deal of self-education wherein prospective students understand that they might not ultimately wind up working in the professional games industry, the interest exhibited in the game design and development programs at RIT is without bounds. Originally planned for an entering freshmen class of 30 students per year, the program now draws approximately 120 students per year out of an application pool of nearly 1,000. The program has had to wait-list applicants and has seen unprecedented growth in early-decision applications as well as applications from geographic areas that have had less representation in more traditional programs.

The interest in domain-centric computing speaks directly to the faculty's desire to match relevant experiences to foundation material. It also speaks to the faculty's desire to create the appropriate connections between traditional computing foundations and what students want to pursue in their academic careers. We feel that being able to draw these connections is appropriate for younger disciplines, such as game design as development, because connecting students to their field of study is important for retention and student ability. During a period where interest in traditional computing programs is declining, the focus on new programs gives educators the chance to experiment with new approaches that have the ability to reach our students. **MM**

Acknowledgments

We thank our students, without whom most of this work would be impossible, and RIT, which has supported this work and the formation of the game design and development degree programs at both the undergraduate and graduate levels. We also thank John Nordlinger, Jamie Cromack, and others at Microsoft Research who have funded several of these initiatives and have been supportive of the notion of using games as a tool for computing education in general across a wide number of institutions and academic endeavors. We thank Mary Flanagan for her partnership and collaborative guidance in our involvement in her work on Values at Play. Additional thanks is given to David Parks at Linden Labs, Luis Ramirez of Vicarious Visions, and Peter Kuhn of RIT for their work with the M.u.p.p.e.t.s. system. Finally, we thank David Luehmann, Chris Satchell, and David Mitchell at Microsoft Games Studios for their general support of our work.

References

1. J.D. Bayliss, ''The Effects of Games in CS1-CS3,'' *J. Game Development,* vol. 2, no. 2, 2007, pp. 7-18.
2. J.D. Bayliss and S. Strout, ''Games as a 'Flavor' of CS1,'' *Sigcse Bulletin,* vol. 38, no. 1, 2006, pp. 500-504.
3. K. Bierre and A. Phelps, ''The Use of MUPPETS in an Introductory Java Programming Course,'' *Proc. 5th Conf. Information Technology Education,* ACM Press, 2004, pp. 122-127.
4. K. Bierre et al., ''Motivating OOP by Blowing Things Up: An Exercise in Cooperation and Competition in an Introductory Java Programming Course,'' *Proc. 37th Sigcse Tech. Symp. Computer Science Education,* ACM Press, 2006, pp. 354-358.
5. C. Egert et al., ''Hello M.u.p.p.e.t.s.: Using a 3D Collaborative Virtual Environment to Motivate

Fundamental Object-Oriented Learning,'' *Proc. Companion to the 21st Ann. ACM Sigplan Conf. Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA), ACM Press, 2006, pp. 881-886.
6. A. Phelps et al., ''MUPPETS: Multi-User Programming Pedagogy for Enhancing Traditional Study,'' *Proc. 4th Conf. Information Technology Education,* ACM Press, 2003, pp. 100-105.
7. A. Phelps et al., ''MUPPETS: Multi-User Programming Pedagogy for Enhancing Traditional Study: An Environment for Both Upper and Lower Division Students,'' *Proc. Frontiers in Education,* ASEE/IEEE Press, 2005, pp. S2H8-S2H15.
8. A. Phelps et al., ''Games First Pedagogy: Using Games and Virtual Worlds to Enhance Programming Education,'' *J. Game Development,* vol. 1, no. 4, 2006, pp. 45-64.
9. A. Phelps et al., ''An Open-Source CVE for Programming Education: A Case Study,'' *Proc. Siggraph Supplemental Notes—Educators Half Day Session,* ACM Press, 2005; http://portal.acm.org/citation.cfm?id=1198555.1198686&coll=ACM&dl=ACM&CFID=43816555&CFTOKEN=52554014.
10. A. Phelps and D. Parks, ''Fun and Games with Multil-Language Development,'' *Queue,* vol. 1, no. 10, 2004, pp. 2-12.

Contact author Andrew M. Phelps at amp@it.rit.edu.

Contact editor Qibin Sun at qibin.sun@ieee.org.