The Incredible Rainbow Spitting Chicken: Teaching Traditional Programming Skills Through Games Programming

Patricia Haden
School of Information Technology and Electrotechnology
Otago Polytechnic
Dunedin, New Zealand
phaden@tekotago.ac.nz

Abstract

Introductory programming courses must provide sound instruction in basic programming skills while still maintaining a high level of student engagement. At Otago Polytechnic we have recently introduced a second year programming course that teaches traditional core topics in the context of "games programming". While building a variety of computer game applications, students study algorithm design, complex data structures, recursion and class architecture. Early experiences with the course have shown it to be both effective and enjoyable. In this paper we present the rationale and structure of the course, and describe some of the course materials.

1 Introduction

Computer programming has always been a challenge to teach successfully. Since "CS101" became a common part of the undergraduate curriculum, educators have struggled to understand why students generally have such difficulty learning to be good programmers (Decker & Hirshfield, 1993; McCracken et. al., 2001; Garner et. al., 2005).

At Otago Polytechnic, we have trialled a number of teaching methodologies and have found that, beyond natural aptitude (over which we have no control), the factor that seems to best predict student success in *introductory* programming is mastery of core programming principles (Haden & Gasson, 2004). This pattern has also been observed by Duke, et. al. (2000). If a student can be given a deep understanding of the principles of basic data manipulation and flow of control, advanced programming esoterica seem to follow naturally.

The inherent difficulty in drilling students in programming basics is that it can be deadly boring. Using if-statements and for-loops to print assorted series of numbers rapidly loses its appeal. Many educators have observed that students who are not engaged in course material are students who probably will not learn (e.g. Kearsley & Shneiderman, 1999; Loblo, et. al. 2000).

The problem of maintaining student engagement is becoming increasingly acute because our incoming students are often members of the "Nintendo Generation" (Guzdial & Soloway, 2000). This group, now entering their late teens, have expectations of how a computer should behave that are based on the World Wide Web, with its elaborate graphical elements, and PC and Platform games, with their rich AI and fast-paced interactivity (cf. Phelps, Bierre & Parks, 2003). Sending "Hello World" to the console is simply not adequate for this population.

We have recently introduced an early programming course that follows the "if you can't beat them, join them" philosophy. Our students are computer game players, so we have decided to teach them programming in the context of building computer games. To this end, we now offer a second year C++ programming paper titled Interactive Worlds: Introduction to Games Programming. We are able to embed all the traditional core programming topics – data structures, flow of control, class architecture, algorithm development – into the design and creation of simple computer games. The course structure also allows some discussion of principles of human-computer interface, a dash of trigonometry, and even an opportunity to consider the social implications of computers in our lives. The students see themselves creating computer games, which is something they've long wanted to do. But in the process, they are reading and writing huge amounts of code, and gaining a firm grounding in the principles they need to become quality programmers.

The success of the first two offerings of the games programming course (in 2004, and currently running in 2005) has been notable. While we have as yet made no attempt to precisely quantify programming skill (see Fincher et. al., 2005, for a discussion of the difficulties in this task), good progress in coding facility has been clear for all students in both years. In some cases, this progress has been dramatic. We have been able to observe directly the perfect attendance, unflagging enthusiasm and willingly committed out-of-class work hours of the course participants. We were also able to observe the very polished games produced by students in the first offering of the course, and the strong programming skills those students brought into their final year of study. Informal code inspection and student feedback (see below) are also extremely positive.

In this paper, we will first discuss the role of games programming in the traditional CS curriculum, and the views of other educators employing this approach. We will then describe the structure of the Interactive Worlds course and present some of the curricular materials used. Finally, we will consider the extent to which Interactive Worlds has been an effective vehicle for improving our students' programming abilities.

2 Games Programming in the CS Curriculum

As the games industry becomes increasingly profitable, a number of CS institutions have introduced some form of Games Programming into their traditional CS curriculum. These courses fall into two broad classes: those intended to prepare students for employment in the games industry (e.g. Parberry, Roden, & Kazemzadeh, M, 2005; Coleman, Krembs, Labouseur, Weir, 2005), and those which, like our course, see games programming as primarily a vehicle for teaching programming skills in a highly motivating, yet pedagogically robust way.

It is generally accepted in the literature that a programming assignment that involves building a real application is more appropriate than one that involves an isolated and unrealistic processing task (e.g. Huang, 2001). Further, for the majority of students, building a computer game is simply more fun than implementing, for example, a mock bank account or billing system (Ross, 2002; Valentine, 2005; Curtis, 2005). As Becker (2001) points out when discussing her experiences using games programming in a first year programming course: "While having fun is not typically high on the list of teaching goals, its value should not be underestimated. Students who are having fun work harder, longer, and are more apt to expand on what is taught than those who simply wish to get it over with and pass the course." Thus by using computer games as programming tasks, one achieves both real world relevance and personal engagement.

Fortunately, this can be done without sacrificing pedagogical appropriateness. Computer games are nontrivial programming entities, which involve many topics from the traditional CS curriculum. These include, but are not limited to: syntax, flow of control, data structures, event-driven programming, human-computer interface design, artificial intelligence, physical simulation, graphics and animation, multimedia design and algorithm design (cf. de Laet, Slattery, Kuffner, & Sweedyk, 2005). Pleva (2004) notes the extensive overlap between principles that can be illustrated via games programming and the ACM recommended guidelines for computer science (ACM, 2005). Additionally, if a games programming course incorporates large projects and group work, students can gain experience in the entire software development process (Jones, 2000).

Not surprisingly, given the potential technical complexity of a complete computer game, many of the courses currently being described in the literature are targeted at senior students (e.g. Jones, 2000; Huang, 2001). Our games programming course is intended for less experienced programmers. Our students generally have only two previous semesters of programming experience – one procedural, and one object-oriented – both taught in Pascal. One great advantage of games as a programming context is that they range naturally in complexity from very simple to nearly unmanageable. We have found that with careful selection of the games to be implemented, and careful coding support where required (see Sections 3 and 4 below) even relatively novice programmers can build working games of which they can be proud.

A particular benefit for our inexperienced programmers is the natural relationship between games and the object-oriented programming paradigm. Prior to entering the Interactive Worlds course, the majority of our students will have had only 12 weeks of instruction in OO design and implementation. Many of them are still struggling with concepts like class architecture, method assignment, inheritance and polymorphism. Many are still not confident of the correct syntax for instantiating objects and writing and calling class methods. Fortunately, it is often especially easy to conceptualise a game in terms of interacting objects, which are then easy to map to an OO architecture (Leska & Rabung, 2005). Our student feedback (see below) indicates that from a student perspective, one of the great benefits of the games programming approach is the extent to which it helps clarify object-oriented theory.

3 Programming Environment

The high graphical load of the modern computer game is both a great advantage, and a great challenge. Giguette (2003) notes that: "…though students have no trouble playing games, they often have trouble programming them. Implementing realistic details, flexible user interfaces, or interesting graphics is beyond the capabilities of many CS1/CS2 students." He suggests that graphical complexity must be sacrificed by using, for example, text-based outputs rather than graphical ones. We agree that the Windows API, OpenGL and Direct X are probably beyond the reach of novice programmers. However, we have found that with the right programming environment, students with even fairly rudimentary programming skills can produce software that is graphically impressive and supports complex user interactivity. In the Interactive Worlds course, all development is done in C++ using Borland's C++ Builder IDE (Borland, 2005). This tool comprises a robust C++ compiler and a "point-and-click" screen painter. Students drop buttons, menus, text boxes, timers and a variety of other built-in components onto a blank form, and then write the code for underlying event handlers such as button click, mouse down, text change, etc. C++ Builder provides a number of powerful graphics classes and components that allow students to produce quite impressive graphics and animations using a set of simple method calls.[2] If our goal in the course was to produce professional games programmers, we would be obligated

[2] The Borland IDE is, in fact, so simple to use that we have successfully employed the Object Pascal version (Borland Delphi) in our first-year programming classes for several years. See Haden & Mann (2003) for details.

to expose students to the more complex tools used in the games programming industry. However, our goal is to provide an educationally sound and motivating programming class, which is best achieved by using an empowering development environment.

4 Course Curriculum

Interactive Worlds is a 17-week course with one lecture and two practical sessions each week. The lecture series concentrates on theoretical issues in game design and construction, while the practical sessions focus on programming and, to a small extent, comparative analysis of existing game software.

We have chosen to follow the incremental development model of Parberry (Parberry, 2001, Parberry, 2002; Parberry, Roden, & Kazemzadeh, M 2005). In this approach, course tasks form a series, with each building upon code developed in previous exercises. At each step, the student is introduced to new practical and theoretical topics. At the end of the series of tasks, students have a number of code modules or classes that they can then use in larger projects. For example, the series of programming tasks our students do in the first seven weeks of the course produce a complete set of classes for implementation and management of animated 2D sprites. Students then use these classes when building their main course project, a 2D side-scroller game.

In 2D side-scrollers, the user controls a main character, which can be steered through a 2D world. The appearance of movement is achieved by scrolling the game background from side to side behind the character – hence the name "side-scroller". Traditionally, the player character encounters other game entities, both helpful and harmful. Well-known examples include Charlie The Duck (www.wieringsoftware.nl), the early Mario games (www.nintendo.com) and the original Duke Nukem (www.3drealms.com). The 2D side-scroller was an extremely active genre in the early days of graphical computer game development. These games can be implemented quite simply, with stick-figure graphics and single axis movement. Alternatively, they can incorporate elaborate artwork, realistic physics and an emotive back-story. (See *Bud Redhead: The Time Chase* (www.space-ewe.com) for an example of a 2D side-scroller that compares favourably to many modern computer games.) As a student project, the side-scroller thus provides a wide range of options, allowing the more adventurous students to extend themselves, while still being tractable for less confident programmers. The games produced by students in the first offering of Interactive Worlds are discussed in more detail in Section 6 below.

The general course plan is shown in Table 1.

Week	Lecture	Practical 1	Practical 2
1	Games History and Genres	C++ vs Pascal	File hierarchy
2	Game Play – It's Not About The Graphics	The Game Event Cycle	Linked Lists
3	Playability	Project 1 - Tetris	Project 1 - Tetris
4	Process and Documentation	Platform Games	Design Document
5	User Interface in Games	Screens, Controls and Sound	Coding – Icon Design
6	2D Artwork And Sprite Animation	Simple Sprite Animation	Directional Sprite Animation
7	2D Background Animation	Simple Tile Map	Scrolling Tile Map
8	Graphics 3: 3D Graphics –	GMAX – 3D Objects	3D Environments
9	Collision Detection	Physics Algorithms	Frogger
10	Trajectories and Gravity	Simple Cannon	Frogger
11	AI 1 – FSM	Approach-Avoidance	Sim Swamp
12	AI 2 - Complex Behaviours	Maze Gen and Solve	Maze Gen and Solve
13	Engines and Mods	Engines 1	Engines 2
14	Enhancing The Experience With Sound	Good Sound, Bad Sound	Project Checkpoint:
15	Windows Programming and DirectX	Windows and DirectX 1	Windows and DirectX Part 2
16	Games and Society	Project Work	Project Work
17	Theory Exam	Project Work	Project Work

Table 1: Course Curriculum

5 Course Materials

Each of the practical coding exercises serves to demonstrate some basic computing principles. In this section, we will describe a selection of these tasks.

5.1 Linked Lists with the Incredible Rainbow Spitting Chicken

During their first year of programming instruction, our students will have had only minimal experience with complex data structures, via arrays and records. In their second year of programming, they need to be introduced to more advanced examples. We use the Incredible Rainbow Spitting Chicken task to teach students to understand and build a linked list object. In this task, students build a small graphical application in which a

chicken[3] (implemented as a primitive image object) can be moved back and forth across the bottom of the screen. When the spacebar is pressed, the chicken "spits" a randomly coloured circle. The circles travel up the screen, and are destroyed when they reach the top of the screen. A screenshot is shown in Figure 1. The coloured circles are managed via a standard linked list object with standard methods Create, AddNode, DeleteNode and CountNode.

Figure 1: The Incredible Rainbow Spitting Chicken

In this practical, students are first given a short lecture on the logic of linked lists and their operations. Then the Incredible Spitting Chicken task is described. We comment on the fact that because the coloured circles are being created and destroyed dynamically in response to user input, it is not convenient to allow at design time for a fixed number of circles. Thus the familiar static array is not an appropriate data structure for this application. Students can see that a linked list, with dynamic addition and deletion of elements, is more efficient.

As this is a very early programming task, students are given a "code skeleton" in which some of the code structure is provided, and they are required to fill in the missing bits. This code skeleton technique is used throughout the course. As the course proceeds, the code skeletons become increasingly sparse until, at the end of the course, students are generating all their own code.

To date, 100% of students have been able to correctly implement the incredible spitting chicken. The resultant application, though technically very simple, is interactive, attractive to look at, and fun to play with. We believe the task is more motivating and engaging than traditional linked-list exercises such as reading in a file of text records and putting them in a linked list, yet still gives students a good first exposure to the theory and practice of complex data structures.

[3] Many of the graphic images used in the Interactive Worlds course are obtained from a wonderful collection of royalty-free graphics available at www.reinerstileset.4players.de:1059. We wish to express our appreciation to the generous creator of this remarkable resource.

5.2 Class Design, Inheritance and Polymorphism with Tetris

The first substantial coding project in Interactive Worlds is an implementation of the classic arcade game Tetris, invented by Alexey Pazhitnov. In Tetris a series of shapes fall from the top of the game screen. Each shape is composed of four connected squares. The user shifts and rotates the shapes as they fall in an attempt to fill complete rows at the base of the game screen.

Because of the nature of the block shapes in Tetris, it is an especially nice illustration of inheritance and polymorphism in OO design. The seven Tetris blocks comprise the seven possible combinations of four adjacent squares as shown in Figure 2.

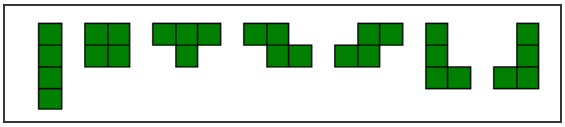

Figure 2: The Seven Tetris Blocks

The blocks can be implemented as a set of four squares with known locations in a grid. Each of the seven block types must have methods to shift left, shift right and shift down. The computation for these methods is identical for all block types (for example, to shift left, subtract one from the x-coordinate of each component square). However, each block type must also rotate, and the process of rotation (i.e. determining the new position of each of the four component squares) is different for each of the different block types. Figure 3 shows a single clockwise rotation for three types of Tetris block: Long-Block, Square-Block and T-Block.

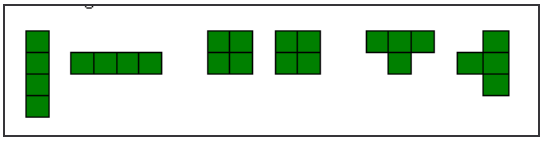

Figure 3: Polymorphic Rotation of Tetris Blocks

Students can easily see that the seven block types share a set of common properties and methods, but implement one of those methods differently – precisely the defining features of a polymorphic architecture. Students cite the Tetris assignment as one that contributes significantly to their programming skill, because of its demonstration of classes and inheritance (see Section 6 below for more discussion of student feedback).

5.3 Physics and Computation with The Pirate Ship of Doom

In the weapon-rich world of games, computation and representation of projectile trajectories is essential. The implementation of realistic physics is required for the accurate shooting needed for game realism. In Interactive Worlds, we introduce these issues via a simple cannon shooting game. A screen shot is shown in Figure 4. This exercise is based on a program found at the DelphiForFun website (www.delphiforfun.com) maintained by Gary

Darby. This site contains a large number of programming exercises that can be used to demonstrate mathematical principles and programming techniques.

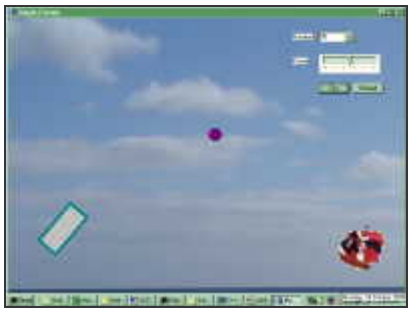

Figure 4: The Pirate Ship of Doom

In the Pirate Ship exercise, basic shapes are used to represent a cannon and a cannon ball. The cannon must be aimed correctly so that the cannonball hits the pirate ship image at the far edge of the computer screen. The vertices of the cannon shape are rotated using standard trigonometric functions for rotation and translation. The initial x and y velocities of the cannonball are computed as a function of the angle of rotation and a user-determined power value. The trajectory is described by moving the cannonball image by its x and y velocities at each game clock cycle. Semi-realistic gravity is modelled by incrementing the y-velocity, but not the x-velocity. See Roman (1999) for a detailed discussion of the involved computations.

5.4 Recursion with a Computer Rat in a Maze

Recursion is traditionally a very challenging concept for new programmers to master. In the Interactive Worlds course we are able to demonstrate recursion in the very concrete context of a maze-following program that uses a standard backtracking algorithm[4]. After a brief discussion of recursion and its use in backtracking, the logic of the maze solving algorithm is presented as follows:

To find a path from start cell to finish cell

- For each neighbour
    - Can we move there?
    - If yes, is there a path from there to the finish cell? (This is the recursive call)
        - If yes, we're done
    - If no, back up and try the next neighbour

For simplicity, the "rat" in this task is implemented by illustrating the sequence of locations traversed by the algorithm. As the program "touches" each square of the maze, it colours it, using different colours for the initial traversal and for the backtrack. This enables students to see the backtracking process clearly. An example is shown in Figure 5, where the red squares are the main path, and the silver squares are backtracking.

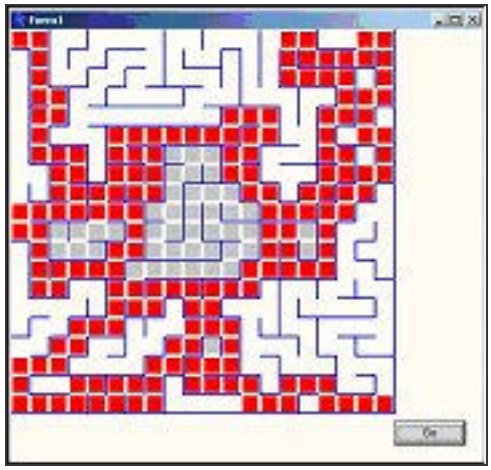

Figure 5: Maze Solver Task Showing Recursive Backtracking Path

6 RESULTS

As stated above, we have made no attempt to formally measure the effect of participation in the Interactive Worlds course on programming skill. Doing so presents a variety of technical difficulties (starting with simply defining programming skill) and ethical issues, as it is not acceptable to deny some students an effective course in order that they may serve as control subjects. We must thus rely on somewhat speculative evidence for our assessment of success. We begin by considering the performance of students on in-class tasks, and performing an informal code inspection of the final 2D side-scrollers produced in 2004. Additionally, we have collected subjective feedback from students currently participating in the second offering of the course, and from a small sample of former students. By all metrics, we feel we have achieved our goals of delivering an enjoyable course while improving student programming skill. We discuss these data in detail below.

6.1 In-Class Performance

When using the Incremental Development method described above, it is essential that students not fall behind in their course work. Since each course unit builds upon previous ones, a student who misses a single task may be poorly prepared for several subsequent sessions. To encourage students to stay up-to-date, most practical session tasks are considered "checkpoints", and students earn marks for completing them. The combined checkpoint tasks total 20% of the student's final course mark. In addition to the exercises detailed in Section 5 above, there are checkpoint tasks for various stages in the construction of the graphics classes, HCI evaluation, simple algorithmic AI, implementation of a finite state machine and experimentation with 3D graphics. During the 2004 offering of the Interactive Worlds course the overall checkpoint task completion rate was 81.67%. This high value indicates that students generally obtained

[4] Students also write an iterative routine for random maze generation in this task. For an example, see http://www.mazeworks.com/mazegen/mazetut/.

sound performance in the programming basics associated with each checkpoint task.

6.2 Code Inspection

As discussed above, the final project in Interactive Worlds is the production of an original 2D side-scroller game. This is a substantial assignment, worth 40% of the total class mark. Students work in groups of two to four. Our experience is that group work allows for the construction of larger and more complex projects than could be produced by a student working alone. Group work also exposes the student to the sometimes complex dynamics of working in a team, which is an important skill in the IT industry (de Laet et. al, 2005). In the first offering of the course, eight final projects were submitted (created by 21 students working in eight groups). An informal code inspection was performed, to check for achievement of important course learning objectives. The results of the inspection are summarised in Table 2.

Learning Objective	Projects demonstrating objective (Total N = 8)
Correct C++ syntax, including appropriate data types, flow of control and use of pointers.	8
Correct division of units into separate .cpp and .h files	7
Correct fielding of user input. Correct assignment of actions to user events	8
Correct implementation and use of linked lists.	7
Correct definition of simple classes	8
Correct use of inheritance and polymorphism as appropriate	5
Creation and implementation of 2D multi-frame sprite animation	8
Correct use of external input files	8
Correct implementation of simple physics -- rectangle collision detection, gravity, trajectory	4
Correct use of simple AI algorithms (e.g. approach/avoid) and/or finite state machine.	5

Table 2: Code Inspection

All student projects showed a strong grasp of C++ syntax and file management, although most students had no previous exposure to the language (two students had previously taken a course in Java; one student was simultaneously enrolled in a course using C#).

While all student projects contained correct simple classes, only 5 of 8 projects correctly used inheritance. The most common error was declaration of multiple independent classes that should have been descendants of a common ancestor.

All projects incorporated the animation classes developed during the term, and were able to extend them to provide the full required functionality needed for backgrounds and animations in their games.

A simple 2D side-scroller can be implemented without physical simulation or artificial intelligence. Nonetheless, half of the projects incorporated physical simulation of some kind (mostly gravity for falling objects), and five of eight used simple AI algorithms that had been presented in class lectures and practicals. One project used a finite state machine to control the behaviour of enemy game entities.

This summary, although informal, indicates that the Interactive Worlds course was successful in teaching our novice programmers a new language and providing them with a number of more advanced programming skills than they had previously acquired. The design of good class structures stands out as an especially difficult problem.

6.3 Student Self-Report Feedback

Nine students from the current offering of Interactive Worlds (65% of the enrolled students) and six students from the first offering (29% of the enrolled students) completed self-report survey forms canvassing their views on the course. Current students were asked to fill out the survey during class. Former students were contacted by e-mail. We had current e-mail addresses for 14 of the 21 students who completed Interactive Worlds in 2004. Of those students, six responded to the e-mail survey. Questions included both Likert-scale ratings, and free comments. Responses are summarised in Tables 3 and 4.

How has participating in IT220 affected your programming skills? They are now:				
Much Worse	**Worse**	**Unchanged**	**Better**	**Much Better**
0	0	0	6	3
IT220 is an enjoyable class.				
Strongly Disagree	**Disagree**	**No Opinion**	**Agree**	**Strongly Agree**
0	0	0	4	5
Games programming is an effective way to teach basic programming techniques.				
Strongly Disagree	**Disagree**	**No Opinion**	**Agree**	**Strongly Agree**
1	0	1	3	3

Table 3: Current Student Survey Responses

Current Students: When asked if their programming skills have improved, and if the class is enjoyable, student

responses are uniformly positive. One student felt that games development is not an effective context for teaching programming. In comments, this student expressed a concern that techniques which would be important in industries other than games were being neglected. This is certainly a legitimate concern. In a general CS program, a variety of programming courses are required to insure that students acquire a sufficiently broad set of skills.

Students were also asked to comment on various aspects of the course. When asked in what specific ways their programming skills had improved, seven of nine respondents mentioned their understanding of object-oriented design. As stated above, games programming lends itself particularly well to the demonstration of OO technique.

When asked what aspects of the course had contributed most to their improved coding skills, six of nine respondents referred to the large quantity of code they had written during the course. As noted above, students who are having fun will put more time and effort into their assignments. These students perceive that such concentrated practice directly contributes to improved programming ability.

When asked what parts of the course they found most enjoyable, four of nine students mentioned the opportunity to write complete applications. Students mentioned the pleasure of "sinking my teeth into a substantial project", "making my own games" and "seeing my games running". Apparently even relatively simple games, perhaps by virtue of their well-defined goals, provide a sense of completion and accomplishment that promotes student motivation and satisfaction.

If you have programmed since IT220, how much did you use the skills and techniques you learned in IT220 in the code you have written?[5]			
Not at all	**A Small Amount**	**A Moderate Amount**	**A Lot**
0	0	2	3

How did participating in IT220 affect your programming skills? It made them:				
Much Worse	**Worse**	**Unchanged**	**Better**	**Much Better**
0	0	0	1	5

IT220 was an enjoyable class:				
Strongly Disagree	**Disagree**	**No Opinion**	**Agree**	**Strongly Agree**
0	0	0	1	5

Games programming is an effective way to teach general programming techniques.				
Strongly Disagree	**Disagree**	**No Opinion**	**Agree**	**Strongly Agree**
0	0	0	2	4

Table 4: Former Student Survey Responses

Former Students: Feedback from former students (see Table 4) was also uniformly positive. Although the sample size is small, the responses are extremely consistent, and lead us to hope that the Interactive Worlds class is, in fact, both enjoyable and effective.

When asked what skills acquired in Interactive Worlds they were using in their current work, students mentioned C++, OO design and linked lists. One student mentioned that he continues to work on his games in his spare time. This is not, in our experience, a comment often heard about old assignments in more traditional programming courses.

6.4 User Feedback

Of course, the ultimate evidence of successful programming is the production of good software products, and by this criterion we feel the Interactive Worlds course has been a great success. Several of the original games produced in the Interactive Worlds course in 2004 were demonstrated at a recent departmental public event, and were received with great enthusiasm. Screenshots of some of the games produced in 2004 are shown in Figure 6.

Figure 6: Sample Screenshots from Interactive Worlds Student Projects

[5] One of the former students does not program in his current job.

7 CONCLUSION

Computer science educators need not be distressed by the attitude of today's computer science students, who think computers are for CounterStrike, not for calculus. On the contrary, we can embrace the graphical and interactive power of new IDEs and development tools, and use these strengths to develop rich and exciting courses without sacrificing pedagogical heft. Every computer science department probably could probably use an Incredible Rainbow Spitting Chicken.

8 References

ACM Curricula Recommendations Website (2005): www.acm.org/education/curricula.html, Accessed 21-Oct-05.

Becker, K. (2001): Teaching with games: the minesweeper and Asteroids experience, *Journal of Computing in Small Colleges*, **17**(2): 22-32.

Borland (2005): www.borland.com, Accessed 21-Oct-05.

Coleman, R., Krembs, , M., Labouseur, , L. and Weir, J. (2005): Game Design & Programming Concentration Within the Computer Science Curriculum. *Proceedings of the 36th SIGCSE technical symposium on Computer science education*, St. Louis, Missouri, USA, 545-550.

Curtis,S.A. (2005): Word Puzzles in Haskell: Interactive Games for Functional Programming Exercises. *Proceedings of the 2005 workshop on Functional and declarative programming in education,* Tallinn, Estonia,15-18.

Decker, R. and Hirshfield, S. (1993): Top-Down Teaching: Object-Oriented Programming in CS1, *Proceedings of the twenty-fourth SIGCSE technical symposium on Computer science education,* Indianapolis, Indiana, USA, 270-273.

De Laet, M., Slattery, M.C., Kuffner, K. and Sweedyk, E. (2005): Computer Games and CS Education: Why and How. *Proceedings of the 36th SIGCSE technical symposium on Computer science education,* St. Louis, Missouri, USA, 256-257.

Duke, R.. Salzman, E., Burmeister, J., Poon, J. and Murray, L. (2000): Teaching Programming To Beginners – Choosing The Language Is Just The First Step, *Proceedings of the Australasian Conference on Computing Education*, Melbourne Australia, 79-86.

Fincher. S., Baker, B., Box, I., Cutts, I., de Raadt, M., Haden, P., Hamer, J., Hamilton, M., Lister, R. Petre, M., Simon, Robins, A., Sutton, K., Tolhurst, D., Tutty, J. (2005): Programmed to succeed?: A multi-national, multi-institutional study of introductory programming courses. *Technical Report 1-05*, Computing Laboratory, University of Kent, Canterbury, Kent, UK.

Garner, S., Haden, P. and Robins, A. (2005): My Program is Correct But it Doesn't Run: A Preliminary Investigation of Novice Programmers' Problems. *Proc. Seventh Australasian Computing Education Conference* , Newcastle, Australia, 173-180.

Giguette, R. (2003): Pre-Games: Games Designed to Introduce CS1 and CS2 Programming Assignments, *Proceedings of the 34th SIGCSE technical symposium on Computer science education*, Reno, Nevada, USA.,288-292

Guzdial, M. and Soloway, E., (2000): Teaching the Nintendo Generation to Program, *Communications of the ACM*, **45**(4): 17-21.

Haden, P and Mann, S. (2003): The Trouble With Teaching Programming, *Proceedings of the NACCQ,*, Palmerston North, New Zealand, 63-70.

Haden, P. & Gasson, J (2004): Easy and Effective Streaming for Introductory Programming Courses,. *Proceedings of the NACCQ*, Christchurch, New Zealand, 281-284.

Huang, T. (2001): Strategy game programming projects. *The Journal of Computing in Small Colleges*, **16**(4) 205-213.

Jones, R. M. (2000): Design and Implementation of Computer Games: A Capstone Course for Undergraduate Computer Science Education. *ACM SIGCSE Bulletin*, **32**(1), 260-264.

Kearsley, G. and Sheniderman, B. (1999): Engagement Theory: A Framework For Technology-Based Teaching And Learning, http://home.sprynet.com/~gkearsley/engage.htm

Loblo, A.F., Baliga, G.R., Bergmann, S. Stone, D., Shar, A. (2000): Using Real-world Objects to Motivate OOP in a CS1 lab. *Journal of Computing in Small Colleges,* **15**(5):144-156

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y., Laxer, C., Thomas, L., Utting, I., & Wilusz, T. (2001): A multinational, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin*, **33**(4):125-140.

Morrison, M. (2003): *Sams Teach Yourself Game Programming in 24 Hours*. Pearson Education, Sams Publishing, Indiana, USA.

Parberry, I. (2000): *Learn Computer Game Programming with DirectX 7.0.* Wordware Publishing.

Parberry, I. (2001): *Introduction to Computer Game Programming with DirectX 8.0*. Wordware Publishing.

Parberry, I, Roden, T. Kazemzadeh, M. (2005): Experience with an Industry-Driven Capstone Course on Game Programming, *Proceedings of the 36th SIGCSE technical symposium on Computer science education.* St. Louis, Missouri, USA, 91-95

Phelps, A., Bierre, K. and Parks, D. M. (2003): MUPPETS: Multi-User Programming Pedagogy for Enhancing Traditional Study. *Proceeding of the 4th conference on Information technology curriculum*, Lafayette, Indiana, 100-105

Pleva,G. (2004): Game programming and the myth of child's play. *Journal of Computing Sciences in Colleges*, **20**(2):125-136.

Leska, C. and Rabung, J.(2005): Refactoring the CS1 course. *Journal of Computing Sciences in Colleges*, **20**(3):6-18

Roman, E. (1999): Gravity FAQ.

http://www.gamedev.net/reference/articles/article694.asp

Ross, J.M. (2002): Guiding Students through Programming Puzzles: Value and Examples of Java Game Assignments. *ACM SIGCSE Bulletin* **34**(4):94-98

Valentine, D (2005), Playing around in the CS curriculum: Reversi as a teaching tool, *Journal of Computing Sciences in Colleges*. **20**(5): 214-222.