Experience with an Industry-Driven Capstone Course on Game Programming

[Extended Abstract]

Ian Parberry
Department of Computer Science & Engineering
University of North Texas
Denton, TX, USA
ian@unt.edu

Timothy Roden
Department of Computer Science & Engineering
University of North Texas
Denton, TX, USA
roden@cs.unt.edu

Max B. Kazemzadeh
School of Visual Arts
University of North Texas
Denton, TX, USA
maxk@unt.edu

ABSTRACT

Game programming classes have been offered at the University of North Texas continuously since 1993. The classes are project based, and feature collaborative coursework with art majors in UNT's School of Visual Arts. We discuss the design that enables them to simultaneously provide both training for students intending employment in the game industry, and a capstone experience for general computer science undergraduates.

Categories and Subject Descriptors

K.3.2 [**Computing Mileux**]: Computers and Education-Computer and Information Science Education[Computer science Education]

General Terms

Design, Experimentation, Measurement

Keywords

Game programming, capstone, undergraduate education

1. INTRODUCTION

In 1993 we introduced a game programming course to the undergraduate computer science program at the University of North Texas. At the time, this was a controversial, much-challenged, and difficult move. There were no course materials, books, or web pages available. Interestingly, the only objections were from faculty — both the students and the administration were in favor of the class. During the first few years the class was offered, objections were raised about the industry-driven focus of the class and the perceived trivial nature of entertainment computing. Since 1993 the initial game programming class has evolved with the fast-moving game industry, and spawned a second, advanced game programming class. After more than a decade of operation, our game programming classes have positioned our alumni for employment in companies including Acclaim Entertainment, Ensemble Studios, Gathering of Developers, Glass Eye, iMagic Online, Ion Storm, Klear Games, NStorm, Origin, Paradigm Entertainment, Ritual, Sony Entertainment, Terminal Reality, and Timegate Studios.

Game programming classes are now gaining acceptance in academia (see, for example, Feldman [6], Moser [10], Adams [1], Faltin [5], Jones [8], Becker [3], Alphonce and Ventura [2], and Sindre, Line, and Valvåg [13]), resulting in a proliferation of new classes and programs nationwide, and a move towards a professionally recommended curriculum in game studies [7]. In contrast to institutions such as Digipen, Full Sail, and SMU's Guildhall that offer specialized degrees or diplomas in game programming, we offer game programming as an option within a traditional computer science curriculum. Keeping in mind that many institutions are starting game programs, and many of them are designing their curricula in an *ad hoc* manner, the purpose of this paper is to share some of what we have learned from experience over the last decade by describing our game programming classes, the design philosophy behind them, and some of the potential pitfalls.

We begin by discussing game industry needs in Section 2, and some important issues in the design of a game programming class in Section 3. We will discuss the introductory class in more detail in Section 4. Finally, in Section 5 we examine the impact of game programming on the computer science program at UNT.

2. WHAT GAME COMPANIES WANT

Game companies want C and C++ programmers with general competence in technical subjects typically found in an undergraduate computer science program such as programming, computer architecture, algorithms, data structures, graphics, networking, artificial intelligence, software engineering, and the prerequisite math and physics classes. In addition, they usually demand evidence of the following skills and experience:

1. Work on a large project, that is, larger than the typical "write a program for a linked list" kind of programs that are typically used as homeworks in programming courses.

SIGCSE'05, February 23–27, 2005, St. Louis, Missouri, USA.

2. Creation of a game demo or two, something nontrivial that plays well and showcases the programmer's ability. This shows that the applicant is devoted enough to have spent their own time to create something, and the perseverance to see it through to completion.
3. That the applicant is a "team player", somebody who can work with other programmers, and just as importantly work with artists and other nontechnical people.
4. That the applicant can learn independently, because the game industry continues to push the boundaries of what can be done using new computer technology.
5. That the applicant is well-versed in game technology, who the important development houses are, and what they are currently reputed to be doing.

While our undergraduates can technically learn enough about the game industry in general and game programming in particular from books to satisfy most of these requirements, our game programming classes are designed to help students achieve them more effectively than they could alone, and encourage them to higher levels of achievement. The requirements listed above are similar to the "Ideal Programmer Qualities" listed by Marc Mencher [9]: self-starters, who possess a team attitude, will follow-through on tasks, can communicate with nonprogrammers, and take responsibility for what they have done.

In addition to satisfying the needs of aspiring game programmers, we quickly found that the game programming classes are attractive to general students as a capstone experience, paralleling the experience of Jones [8]. Indeed, our class projects meet most of the requirements of the capstone project CS390 in [4]. Other employers are also attracted to students who have experience with a group software project with nontechnical partners. Feedback has suggested that game demos created with artists tend to show better in interviews than the typical project created by programming students.

3. DESIGNING A GAME PROGRAMMING CLASS

There are a number of key decisions in the design of a game programming class that affect the outcome in a fundamental way:

1. Should the classes be theory based, or project based?
2. What software tools should be used?
3. Where do programming students find art assets?
4. Should students be free to design any game in any genre, or should their choices be limited?
5. Should students write their own game engine, or work with a pre-existing engine?

On the first question, the options were either a theory class with homeworks and exams, perhaps augmented with small programming projects, versus a project class in which the grade is primarily for a large project programmed in groups. We chose the project option, understanding that students would come out of the classes with two substantial game demos that will play a major role in their first job interview in the game industry.

On the second question, the Computer Science department at UNT was until recently almost exclusively Unix based, with `g++` being the compiler of choice and graphics programming taught using OpenGL. We chose to use Windows, Visual C++, and Microsoft DirectX instead, for two reasons: for those students bound for the game industry it makes sense to expose them to tools actually in use in a significant segment of the industry, and for the rest, it is advantageous to expose them to a different set of software tools before graduation (both of which are encouraged in Section 10.2.2 of [4]).

On the third question, that of art assets, the obvious choice is to have students take advantage of the free art on the web. Our experience is that students benefit substantially from working with art students. We will describe more of our collaboration with the School of Visual Arts at the University of North Texas in Section 4.

On the fourth question, on whether students should be allowed to design and implement a game in any genre, our experience is similar to that of Sindre, Line, and Valvåg [13]. Constraints on the type of game being created (as in [1, 2, 3, 5, 6, 8]) may seem attractive from a managerial point of view because, for example:

- It allows for a more shared experience, enabling students to learn and collaborate across group lines.
- It gives the flexibility to reassign group membership in response to late drops and overheated group dynamics.
- It allows the art class to streamline their process by using a pipeline art production line where necessary.

However, we have found that the element of creativity, student morale, the quality of the resulting games, and the outcomes all suffer when any kind of constraint is placed on the game being developed.

On the fifth and final question, whether to teach with a pre-existing game engine and tools or to have the students create their own custom game engines, the pre-existing game engine option may seem the most attractive at first for several reasons:

- It allows the students to "stand on the shoulders of giants", that is, to achieve more than they can on their own by leveraging existing code.
- It prepares them for the game industry, where they will likely find themselves working on an existing engine, or at least with an existing code base.
- It is easier for faculty to teach from an existing game engine than to teach students to create their own game engines.

However, we have found that the arguments for not using a pre-existing game engine are more compelling in practice:

- Teaching students to use a single game engine simply trains them in its use. The learning curve in a single 15-week class is typically so steep that they run the risk of spending their time wrestling with code rather than developing general skills.
- Existing game engines for educational use tend to be poorly documented, low in features, and unstable. Students find that they spend most of their time trying to force a recalcitrant engine to do what they want it to do, or coding around obscure bugs. They are often resentful of the fact that their grade depends on somebody else's ability to write code, particularly when it is obvious that "somebody else" writes bad code.

- The code for existing game engines is generally *production code*, code that is designed to run fast and be maintainable, rather than *teaching code*, which is further designed to teach basic concepts.
- Students who write their own game engines get first-hand experience with their internal workings, and are thus able to more quickly pick up the details of the proprietary game engine at their first job.
- Students entering the game industry will most likely spend the majority of their professional lives modifying and making additions to somebody else's code. This is the last opportunity that they will have to devote major slices of their time on their own game engine.

For these reasons, we opted to teach game engine programming with the class project being to create a game engine using some standard utilities, rather than modifying a free or proprietary game engine.

4. THE INTRODUCTORY GAME PROGRAMMING CLASS

The introductory game programming class was introduced in 1993 as a special topics class. Despite some initial resistance from faculty, it received its own course code CSCI 4050 and catalog entry in 1997, effective in Fall 1998. It is offered once a year in Fall semesters.

CSCI 4050 started out in 1993 as a 2D game programming class for DOS, changed to DirectX 3, and has been updated annually to keep pace with each new release of DirectX, from DirectX 5–9. Recently, elementary 3D techniques for a simple billboard game has been introduced to the curriculum. CSCI 4050 is a project class. Students must attend lectures, but the final grade is for a game programmed in teams. To make this as real-world as possible, the students are given an ill-defined objective, as recommended in Sections 10.3.2 and 10.4 of [4]. In the first class meeting, the students are shown a slide that describes the grading system as follows:

A:	it really knocks my socks off
B:	it's a pretty cool game
C:	it's an OK game
D:	it's not there, but at least you tried
F:	you really blew it off, didn't you?

Two kinds of points are awarded: completeness points and techno points. Completeness points are awarded for things such as:

- Does it run without crashing?
- Are there few (preferably no) bugs?
- Does it have an intro, a title screen, a credits screen, a menu screen, help screens?
- Does it play with the keyboard, mouse, and/or joystick?
- Does it have sound support?
- How is the game play? Is it fun?

Techno points are awarded for implementing technology not covered in class. Examples include, but are not limited to:

- MP3 instead of WAV format sounds
- Showing video clips using DirectShow
- Lighting effects (eg. directional light, sunset, shadows, lense flare)
- Pixel and vertex shaders
- Network play using TCP/UDP/DirectPlay

The students in CSCI 4050 are usually seniors in the computer science program, who are technologically savvy and experienced programmers. They are usually quite capable of reading the DirectX documentation themselves. For them, the biggest road-block is picking the small subset of techniques that they actually need from the wealth of options available. The lectures in CSCI 4050 focus on getting started, and leave exploration of options in the more than capable hands of the students.

The first author has developed a novel teaching technique called *incremental development*. Rather than going through the DirectX documentation in detail, we teach using a basic game called *Ned's Turkey Farm*, a simple side-scroller in which the player pilots a biplane and shoots crows. The aim is not to teach this game *per se*, but rather to teach the development of games in general using this engine as an example. It is designed to have many of the features of a full game in prototype form so that students can, if they wish, use code fragments from it as a foundation on which to build their own enhancements. Earlier versions of the code and lecture notes have been published in book form (Parberry [11, 12]).

During lectures we have a laptop with 3D acceleration and an overhead projector available in the classroom. The laptop is set up as a game development platform, with Visual C++ and the DirectX SDK. It is important to be able to show and manipulate the code in class, rather than just show a pre-prepared slideshow. This hands-on attitude to the code in class helps us avoid a disconnect between the code and the lectures: in many classes the code and the lecture material seem to have very little intersection.

The code is currently organized into a sequence of 11 demos. Each demo is built on top of its predecessor. A file difference application, such as `windiff` is used in class to highlight the changes in code that must be made to add the new features. An average of one demo is presented per week. A typical class begins by running the demo and pointing out the new features, followed by a powerpoint slideshow describing the new demo, its new features, the theory or principles behind them, and any implementation details, but at a high level without getting bogged down in the code. This is followed by running `windiff` and going through the code changes in more or less detail depending on the complexity and difficulty of the code. Often, we run Visual C++ to show students in real time the effects of minor code tweaks.

CSCI 4050 is taught in parallel with a game art class taught to art students in the School of Visual Arts at UNT. Part of the art students' grade is to produce the art work for a game programmed by the students in CSCI 4050. To encourage group synergy we teach both the art and programming classes at the same time in different rooms in the same building. Classes run for 3 hours in the evening, and the final hour is reserved for group meetings between the artists and programmers. We have experimented with running the classes at different times, and at the same time in different buildings, resulting in both cases in a massive drop-off in meeting attendance, and a corresponding decrease in the quality and number of completed games at the end of the semester.

Allowing students to form their own groups based on common interests has proved to be the best way of maintaining

interest and excitement about the projects. At the end of the first class we take the students in both classes — typically 30–35 programmers and 15–20 artists — into a large classroom and have them stand up sequentially and introduce themselves to the class, asking them specifically to talk about what kind of games they like to play, what kind of game they would like to create, and any prior experience. We then allow them to wander around at random, and come to the front of the room when they have formed a group of two programmers with one artist. We have found that the amount of artwork required by a simple sprite game is within the ability of a single art student to create in a single class. However, we always have one or two groups of odd sizes, which are handled in a case-by-case manner.

The final projects in CSCI 4050 are presented to the instructor in a series of 30-minute slots over two days during Finals week. They are graded on the final executable only, the instructor does not look at source code. After demonstrating the game and allowing the instructor to play, the students are quizzed on their individual contributions to the game, to ensure that they actually did what they claimed to have done. Grading on the executable only is a radical departure from other classes that the students have taken in the computer science curriculum, but is an important real-world constraint.

Starting in the Fall 2002, we instituted a game contest for students in CSCI 4050 and the associated game art class. Entry is strictly optional, and does not contribute to grades. The contest is judged by a panel of 4 or 5 local representatives from the game industry. Prizes are donated by Texas game and publishing industries, ranging from the more expensive books and games to less expensive T-shirts and posters. The contest lasts 2–3 hours, and is open to the general public.

Holding the contest in the final week of classes, approximately one week before the deadline for turn-in of the final projects, encourages students to start coding early. Previous attempts at getting students to get started early were focussed on checkpoints and documentation. Preliminary progress reports and play testing dates proved to be positive up to a certain point, after which insistence on more checkpoints and documentation took up valuable time that could more profitably be spent creating the actual game. The game contest is a much more positive way of reinforcing the final deadline.

Our proximity to the DFW metroplex with its high density of game development companies makes it easy to attract guest lecturers. We encourage visits by teams from development houses including artists, programmers, and designers, and have them speak to the combined class of artists and programmers. Rather than technical presentations, we have guest lecturers speak about what it is like to work in the game industry, what it takes to get their first job, and what educational paths the students should pursue. Typically, we have two or three presentations per semester.

5. ENROLMENT TRENDS

We believe that the game programming classes at UNT have had a significant effect on student enrolment and retention. Student numbers are currently dropping in computer science and engineering programs nationwide, which is mirrored at UNT (see Figure 1). Figure 2 shows enrollment figures for the introductory and advanced game programming classes from Fall 1997 to Spring 2004 compared to the total number of Bachelor's degrees awarded by the Department of Computer Science and Engineering. The introductory class was capped at 35 students in 2002. We see that a substantial fraction — between one-third and one-half – of departmental graduates have taken the introductory game programming class.

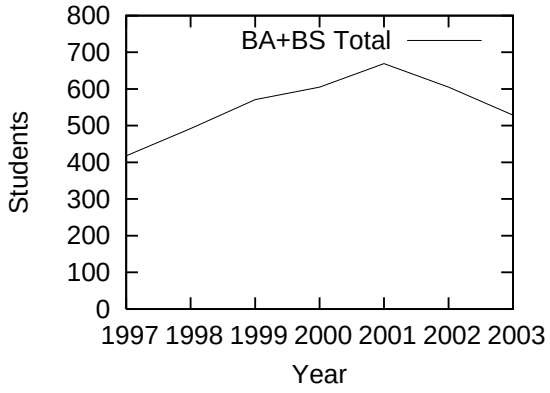

Figure 1: Total enrollments in BA and BS degrees in CSE, 1997-2003.

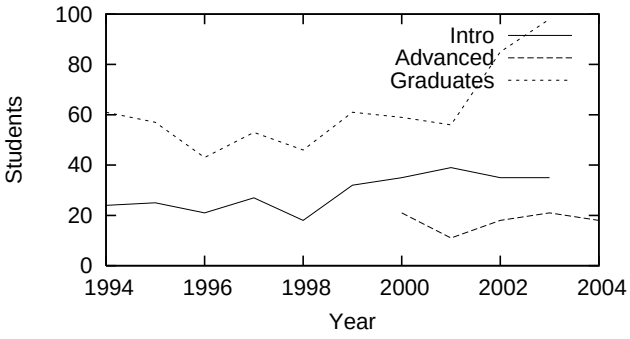

Figure 2: Game class enrollment versus number of CSE Bachelor's degrees awarded. Degrees are listed by academic year, for example, the 2003 figure lists graduation in Fall 2002, Spring 2003, and Summer 2003.

Table 1 shows the results of a survey of students in the undergraduate computer science and engineering programs at UNT in Fall 2003. A total of 197 students were polled in three classes, the two-course freshman-year C++ programming sequence CSCI 1110, CSCI 1120, and the junior-year Data Structures prerequisite to the intro game programming course, CSCI 3400. Despite the total absence of an advertising budget, 79% of students had heard of the game programming classes, and about half of those had heard of them before coming to UNT. We can see future demand for game programming classes, for example, that 49% of students polled intend to take the intro game programming class, and a further 32% say that they may take it. We can also see the effect on the makeup of the undergraduate population, for example, it was a factor in choosing UNT for

Response	1110	1120	3400	Total
When did you hear about the game programming classes?				
Before coming to UNT	47%	40%	27%	41%
While at UNT	33%	33%	55%	38%
Only during survey	20%	26%	16%	21%
Do you plan to take the intro game programming class?				
Yes	50%	40%	59%	49%
Maybe	32%	35%	27%	32%
No	18%	25%	14%	19%
Did they influence your decision to come to UNT?				
The only reason for choosing UNT	6%	4%	5%	5%
Major reason for choosing UNT	19%	12%	5%	14%
Minor reason for choosing UNT	13%	21%	27%	18%
No influence on choice of UNT	63%	63%	64%	63%
Number of respondents	96	57	44	197

Table 1: Responses to Fall 2003 survey. Columns list percentages for the two-course introductory C++ programming sequence CSCI 1110, CSCI 1120, and the Data Structures prerequisite to the intro game programming course, CSCI 3400.

37% of them.

6. CONCLUSIONS

We have had great success over the last decade with a two-course sequence in game programming in a traditional computer science undergraduate curriculum. The classes are project based, and feature collaborative work with art students in the School of Visual Arts. In addition to training aspiring students for the game industry, the classes also provide a capstone style project experience for all computer science students.

Our experience with game industry involvement is that while companies are, with a few notable exceptions, reluctant to provide any sort of concrete support for game development programs in academia, individuals are much more positive. Requests for guest lecturers from industry almost always results in great presentations from motivated, knowledgable, and experienced game programmers and artists.

7. REFERENCES

[1] J. C. Adams. Chance-It: An object-oriented capstone project for CS-1. In *Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education*, pages 10–14. ACM Press, 1998.

[2] C. Alphonce and P. Ventura. Object orientation in CS1-CS2 by design. In *Proceedings of the 7th Annual Conference on Innovation and Technology in Computer Science Education*, pages 70–74. ACM Press, 2002.

[3] K. Becker. Teaching with games: The minesweeper and asteroids experience. *The Journal of Computing in Small Colleges*, 17(2):23–33, 2001.

[4] Computing Curricula 2001: Computer Science. Steelman draft, The Joint Task Force on Computing Curricula, IEEE Computer Society, ACM, 2001.

[5] N. Faltin. Designing courseware on algorithms for active learning with virtual board games. In *Proceedings of the 4th Annual Conference on Innovation and Technology in Computer Science Education*, pages 135–138. ACM Press, 1999.

[6] T. J. Feldman and J. D. Zelenski. The quest for excellence in designing CS1/CS2 assignments. In *Proceedings of the 27th SIGCSE Technical Symposium on Computer Science Education*, pages 319–323. ACM Press, 1996.

[7] IGDA Curriculum Framework. Report Version 2.3 Beta, International Game Developer's Association, 2003.

[8] R. M. Jones. Design and implementation of computer games: A capstone course for undergraduate computer science education. In *Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education*, pages 260–264. ACM Press, 2000.

[9] M. Mencher. *Get in the Game!* New Riders Publishing, 2003.

[10] R. Moser. A fantasy adventure game as a learning environment: Why learning to program is so difficult and what can be done about it. In *Proceedings of the 2nd Conference on Integrating Technology into Computer Science Education*, pages 114–116. ACM Press, 1997.

[11] I. Parberry. *Learn Computer Game Programming with DirectX 7.0.* Wordware Publishing, 2000.

[12] I. Parberry. *Introduction to Computer Game Programming with DirectX 8.0.* Wordware Publishing, 2001.

[13] G. Sindre, S. Line, and O. V. Valvåg. Positive experiences with an open project assignment in an introductory programming course. In *Proceedings of the 25th International Conference on Software Engineering*, pages 608–613. ACM Press, 2003.