The Art and Science of Game Programming

[Extended Abstract]

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

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

Timothy Roden
Center for Advanced Computer Studies
University of Louisiana at Lafayette
Lafayette, LA, USA
troden@cacs.louisiana.edu

ABSTRACT

The University of North Texas has for many years offered classes in game programming to Computer Science students and classes in game art and design to art students. A key feature of these classes is the opportunity for these diverse communities of students to collaborate on joint projects. We describe the features that make these classes unique.

Categories and Subject Descriptors

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

General Terms

Design, Experimentation

Keywords

Game programming, graphics, 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 difficult task, both because there were no course materials, books, or web pages available, and because the industry-driven focus of the class and the perceived trivial nature of entertainment computing made the subject matter controversial. Interestingly, the objections came from faculty - both the students and the administration were in favor of the class. 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.

Despite a rocky beginning, game programming is now gaining acceptance in academia (see, for example, Adams [1], Becker [2], Faltin [4], Feldman and Zelenski [5], Jones [7], Moser [8], and Sindre, Line, and Valvg [11]), resulting in a proliferation of new classes and programs both internationally and nationwide and a move towards a professionally recommended curriculum in game studies [6]. In contrast to institutions such as Digipen, Full Sail, and SMU's Guildhall that offer specialized degrees or diplomas in game programming, UNT offers game programming as an option within a traditional computer science curriculum, and game art and design as an option in their progressive art curriculum. A key characteristic of our educational approach is to bring together computer science and art students in joint projects that enhance the educational experience for both types of student, while grading both types independently according to the metrics of their particular discipline.

2D and 3D Game Design is presently offered to School of Visual Arts undergraduates and graduates as a Junior-Senior-Graduate elective in collaboration with the game programming course in the computer science department. With each new semester the art based game design course logistical challenges are similar to that of the game programming course primarily due to the fact that the game design course is a technology driven course. With each year new game design books and software must be investigated and the existing course curriculum must be updated and redefined to keep up with evolving technology and the technical processes used in the game industry. With each new software version release comes new improvements with usability as well as new formats for output, which require research and testing so as to seamlessly integrate the art student's 2D and 3D assets with interfaces used in the game programming course.

UNT's School of Visual Arts (SOVA) is a progressive conceptual art school that successfully challenges the norms and traditions of present day studio art praxes. SOVA is divided into commercial design, studio art, and educational practice. There is presently no degree in electronic media art or gaming, however, the integration of game design into the SOVA course structure is being carried out with enthusiastic support from both the faculty and the administration.

SIGCSE'06, March 1–5, 2006, Houston, Texas, USA.

Course offerings relating to technology are rapidly expanding and the school is hiring new technologically savvy professors who will teach courses relating to art and technology. A new curricular structure for art and technology is forming at SOVA that references characteristics of innovative interactive art programs such as Carnegie Mellon's Trans-Media Program, NYU's Interactive Telecommunications Program, Parsons School of Design's Design and Technology Program, and MIT's Media Lab.

2D/3D Game Design is presently supported by SOVA electronic media art course offerings that train students in the areas of interface design, web-based art and interactivity, broadcast motion graphics and video-effects, animation, and site-specific digital installation or physical computing.

Game design is an iterative process of defining a set of rules, prototyping a system for interaction, and building game characters, environments, and animations necessary to an interactive narrative. With a broader definition of game design, art and gaming programs can maintain the necessary flexibility to adapt to the evolving industry environment, while providing a stimulating environment to explore and discover new possibilities of what a game can be.

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 and game design classes, the philosophy behind them, and some of the potential pitfalls.

2. INTRO GAME PROGRAMMING

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 and catalog entry in 1997, effective in Fall 1998. It is offered once a year in Fall semesters.

The students in the intro game programming class 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 focus on getting started, and leave exploration of options in the more than capable hands of the students.

We have used a teaching technique called *incremental development*. Rather than going through the DirectX documentation, 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 (see Figure 1). 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 use code fragments from it as a foundation on which to build their own enhancements. The students are graded on the basis of a project, which is to create a sprite-based game in groups together with art students from the concurrent game art and design class.

The code is currently organized into a sequence of eleven code 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.

Figure 1: Screen shot of *Ned's Turkey Farm*.

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.

The code for Fall 2004 and Fall 2005 was organized into twelve incremental demos as follows:

Demo 0, Getting Started: Demo 0 is, for many students, their first Windows application. It fires up a black fullscreen window with a text prompt and waits for user to hit the ESC key to exit. Students learn how to register a fullscreen window, create it, draw graphics on it using the Windows GDI, respond to user keyboard input, and shut down the application gracefully. They are introduced to the concepts of Windows messaging and the message pump.

Demo 1, Introduction to Direct3D: Demo 1 is our first Direct3D application. It starts Direct3D and displays a shoebox background consisting of a floor and a backdrop. It may be run either fullscreen or windowed by changing a global variable.

Demo 2, Scripting and Debugging: The executable for Demo 2 looks the same as Demo 1, but there is a lot more under the hood. We add XML scripting using TinyXML. Now the students can change the behaviour of the demo by editing `Ned.xml` in an XML editor instead of having to recompile the code. This allows the art students to change some of the settings in the game easily.

Debugging may seem like a moot issue until students seriously start creating their own game. The problem with DirectX fullscreen debugging is that DirectX takes complete control of the screen. The Visual Studio debugger is the first line of defense, but some bugs will crash the debugger. The debug code in Demo 2 will let the programmer read debug output in real time in a client console application on a second monitor, or on the screen of a second computer. It will also save the debug output in a file, and display it in the Visual Studio debugger's Output window. It accepts printf-like parameters, and it can be disabled with two keystrokes by commenting out a single `#define`.

Demo 3, The Sprite: Demo 3 is our first attempt at simple real-time animation. A plane sprite moves across the

background. F1 tabs between game view and eagle-eye view, in which the camera pulls back so the programmer can see what is happening "behind the scenes".

Demo 4, Managing Objects: Demo 4 has more objects, managed by a sprite manager and an object manager. The object manager is a first draft only in that it can create objects but not yet delete them. Sprites are now animated with multiple frames of animation. There is now a continuous, infinite scrolling background with the camera in motion to keep the plane in the center of the screen.

Demo 5, Artificial Intelligence: In Demo 5, crows now have simple rule-based artificial intelligence with some randomness thrown in to make them behave slightly differently. They try to avoid the plane as much as they can given a limited attention span. Flocking can be seen as emergent behaviour. The plane fires a bullet when the player hits the space bar. Bullets have a fixed lifetime. When a bullet hits a crow, the crow explodes and turns into a falling corpse, which disappears when it hits the ground. The object manager now has full functionality, in that it can now delete objects and recycle their space.

Demo 6, The Game Shell: In Demo 6 there is a game shell wrapped around the game engine, consisting of an intro sequence (a logo screen, a title screen, and a credits screen), and a menu screen. The player can click out of any of the intro screens. From the menu screen one can play a game or quit by pressing the appropriate key on the keyboard. After each game, the player is returned to the menu screen after they kill the last crow, or pre-emptively by pressing ESC. From there the player can re-enter the game engine. We show how to gain direct access to the back buffer to blit the intro screens there directly.

Demo 7, Sound: Demo 7 introduces sound, managed by a sound manager class. Since DirectSound will not allow a sound to be played multiple times simultaneously, the sound manager keeps multiple copies of each sound, sharing the sound data, and automatically selects the first copy that is not currently playing. The plane engine sound loops.

Demo 8, DirectInput: Demo 8 uses DirectInput to give faster access to input hardware. We start by using it for the keyboard and mouse instead of using the Windows messages like in previous demos. We also add some 2D animation for clickable buttons on the menu page. The mouse is used to press menu screen buttons, as a joystick, and to fire the gun. The buttons on the main menu are drawn as 2D sprites. There is a custom DirectInput mouse cursor.

Demo 9, The Joystick: Demo 9 adds joystick control to the DirectInput code, and adds a device selection screen with radio buttons.

Demo 10, Onscreen Text: Demo 10 adds more complexity to the game, and introduces the drawing of text in screen space. We add multiple levels, with more crows as level number increases, and a success screen in between levels. Player can now be hit by crows, which reduces health and ultimately kills the player. The player's health and number of lives are managed by a score manager. We add text on the screen showing the level number, number of crows, health, lives and score.

Demo 11, Persistence: Demo 11 stores in the Windows registry the game settings that should persist from one execution of the game to the next. We start with the high score list and the initial input device. Checksums are used to detect tampering. New code is added to display the high score list, enter a new name typed in by the player into high score list, and manage the stored high score list.

3. ADVANCED GAME PROGRAMMING

The advanced game programming class was introduced in 2000 as a special topics class. It received its own course code and catalog entry in 2003, effective in Fall 2004. It is offered once a year in Spring semesters. The introductory game programming class is the sole prerequisite.

Advanced Game Programming covers real-time 3D animation. The majority of the grade for the class is for a 3D game created in groups, a typical group consisting of two programming students and two art students from the concurrent game art class taught in the School of Visual Arts. An increase in the number of art students per group over the introductory classes is required because of the increase in work needed to produce 3D models. Programming students are also permitted to use art work from the web, but this has a number of disadvantages, including:

1. Quality: Models often have inappropriate triangle count (too high or too low) and topological defects (degenerate, detached, or sliver triangles, for example).

2. Post-processing: Models often require significant post-processing, for example, they may not be located at the origin, and may have triangles listed in the wrong order for back-face culling.

3. File format: Models are often posted in various formats, for which loaders must be written or adapted from other code. File format converters exist, but they are typically expensive, buggy, or produce low-quality results that require post-processing.

4. Motivation: Programming students are more excited about their games, and hence better motivated, if they can have custom art created on-the-fly in response to group design decisions. Our experience has shown that downloading art from the web typically results in dissatisfied students and lackluster games.

The biggest drawback to having students create a custom 3D game engine is the amount of work involved. It is imperative that students use some basic utilities, for example, the D3DXUtil library provided in the DirectX SDK. We have used a set of improved utilities (including a basic rendering engine, a model importer, and implementations of Euler angles, matrices, vectors, quaternions, axially aligned bounding boxes), published in Dunn and Parberry [3]. Students who wish a different experience are permitted to download and use any free game engine.

As with the introductory game programming class, one of the key elements of this class is the excitement generated by having programming students work with art students. In the advanced game programming class, however, there is a significant barrier to communication between the artists and the programmers: the model file format problem. The art students can work with sophisticated 3D modeling tools such as Maya, Lightwave, and 3D Studio Max, but unfortunately the native file formats generated by these programs are proprietary and difficult to load. The programming students work with Microsoft DirectX, which has strong support for its own file format, called ".x". We have found

Figure 2: Screen shot of *Ned's Turkey Farm* 3D.

this disconnect between the file formats used by the artists and the programmers the most difficult gulf to bridge. All of the plug-ins, exporters, and file converters we have tried have the disadvantage of being expensive or unreliable, or both. Worse still, they fail annually when SOVA upgrades its subscription to the 3D modeling tool of choice, leading to a last-minute scramble to provide software tools in time.

Our current solution is to use the S3D file format proposed in Dunn and Parberry [3] which has the advantage of being a text format, so that programmers can open the files with a text editor to check first hand for errors such as mangled texture file path names, model origin, and scale. The file format, C++ code for a model reader, and plug-ins for Maya and 3D Studio Max are available online.

As with Intro Game Programming, we bring 2 or 3 guest lecturers from the game industry to class, but this time the focus is on technical material. In addition, we have found that requiring students to give a technical presentation to class on an advanced topic of their choice is a way to promote independent learning. In Spring 2006, we plan to teach the class using a 3D version of Ned's Turkey Farm (see Figure 2) currently under development [9].

4. GAME DESIGN AND ART

In addition to becoming proficient in the software taught, each student in the game art and design classes use the series of game design steps in Crawford [5] as a blueprint for how to prepare, plan, and implement a fully functional, playable game by the end of the semester. These steps include: (1) Choosing a Goal and a Topic, (2) Research and Preparation, (3) Design Phase, (4) I/O Structure, (5) Game Structure, (6) Evaluation of the Design, (7) Pre-Programming Phase, (8) Play-testing, and (9) Post-Mortem.

Parberry, Roden, and Kazemzadeh [10] list the following key decisions in the development 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?

The following discusses the impact of these decisions on the Game Design and Art class.

In reference to the first question, the options were a single project course in which the grade is primarily for a large project designed in groups, versus a supplemental projects course where the instructor teaches hands on implementation techniques daily, have regular spaced out small projects, an optional mid-term covering terms and techniques, and the final project. We chose the latter supplemental projects course due to the fact that most students were new to technology and needed to be given smaller organized projects to encourage the practice of techniques covered in class.

On the second question, the 2D and 3D courses require that different software be used. For the 2D course, we use Adobe Photoshop, Adobe Imageready, Adobe Illustrator, Painter programs, Macromedia Flash, and a small amount of Director. We chose to use Painter, Photoshop, and Illustrator to develop the imagery (characters, props, environments), Adobe Imageready to implement/prototype animations and to optimize imagery for output, Macromedia Flash to create a working prototype of the game with animations and behaviors, and Macromedia Director to study the basics of how object oriented programming works so as to provide art students with a better understanding of what programmers face.

At the beginning of the semester, we have a non-digital prototyping workshop to practice working with a set of materials, defining a set of rules, and creating a system of play, to stimulate students to begin thinking about game-play. For the 3D course, we use Maya software, Painter, Adobe Illustrator, and Adobe Photoshop for textures, and Adobe Imageready, Premier, and After Effects for prototyping animations. In the future, we plan to practice creating a MOD version of a game so that students are able to see their models/animations in the 3D environment.

On the fourth question, whether students should be allowed to design and implement a game in any genre, we have discovered the same outcome to be true as in the game-programming course. Restricting the student in any way seems to inhibit development and timely completion. When students collaborate with their teams on a gaming concept, there is a degree of ownership that they have for their idea. They seem to be more excited about their project, and therefore more willing to carry it through to completion.

On the fifth and final question, while 2D and 3D art students are not taught to write their own game engine, they are encouraged to implement a game prototype of some kind on their own within the course of the semester. It gives them a rough sense of how potential characteristics of the completed game will look and behave, and will allow artists to make necessary changes in scale, animation, and tweaks in their model. It also becomes a tool for artists to communicate desired effects to their programming team members.

5. THE LABORATORY

Providing a dedicated game programming laboratory proved to be an important requirement, since the standard open laboratories provided by UNT were unsuitable for the game programming classes for several reasons.

- The bureaucratic process of updating software was slow and cumbersome, since our open labs catered to a wide range of students from the sciences and liberal arts.

- The graphics and sound cards were not up to standard.
- Students developing and playing games proved distracting to other lab users, and game development students soon ran afoul of rules against game playing.
- Since game students are required to work in teams with other programmers and artists, a substantial amount of team meetings debugging need to be actually at the keyboard. The open labs are designed for students who work alone, and in general have a policy of silence.
- Game development students are excited about what they do, and in consequence tend to be rowdy.
- A dedicated game development space provides a place where students can meet and work with other students who share their interest. The area becomes a crucible for independent learning and experimentation that inspires students to greater efforts and achievements.

We started with a small room with three computers in 1993. As space became available, we moved to larger quarters in 1994 and 2001, finally moving into the current location in which we have 570 square feet, 15 computer workstations, and a file server. We opted for computers built and maintained by the university, which provides free, fast onsite maintenance. The computers are on a 2-year upgrade cycle, as opposed to the standard 4-year upgrade cycle, paid for by course fees from the game programming classes. The current computers are 3GHz Intel Pentium 4, with Nvidia 6800 Ultra video cards and dual monitors.

We found that the best organization for the lab is the "bull pit" model, with computers around the outside of the room. The center of the room has tables arranged for conferences and meetings, and for laptops that use the building's wireless network. The computer workstations are connected to the wired network through the fileserver, which serves as a firewall to prevent packet floods generated by rookie game programmers from swamping the building's network.

To foster interaction between students and provide an inviting club-like atmosphere, one corner of the room has been set up with a sofa, a TV, and several game consoles. A typical day will find people playing a game on the console, playing networked PC games, writing code for their own game engines, and engaged in vigorous discussions on subjects ranging from linear algebra to graphics using one of the four large whiteboards.

Finally, the location of the lab is important. Currently it is across the hall from the first author's office, which since the office and lab doors are usually open, encourages interaction between faculty and students. The space is located away from the other faculty offices where the noise generated by the students will not cause a problem for more sober and sensitive colleagues.

To ensure that the lab software is kept up-to-date and that the hardware does not get stolen, we hire a student as lab monitor. He or she is required to keep the lab open for 20 hours per week. The job usually goes to one of the alumni of the game programming courses so that he or she can provide help to the current crop of students. In addition to this paid position, several trusted students also have lab access on the understanding that they provide additional informal open hours by allowing other students to use the lab while they are there. The lab door is fitted with an electronic card-swipe lock that monitors and records entry.

6. CONCLUSION

We have had great success over the last decade with a two-course sequence in game programming in a traditional computer science undergraduate curriculum, and a two-course sequence in game art and design in the School of Visual Arts. The classes are project based, and feature cross-disciplinary collaboration.

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

[3] F. Dunn and I. Parberry. *3D Math Primer for Graphics and Game Development*. Wordware Publishing, 2002.

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

[9] I. Parberry, E. Carson, J. Nunn, and J. Scheinberg. SAGE: A simple academic game engine. `http://larc.csci.unt.edu/sage/`, 2005.

[10] I. Parberry, T. Roden, and M. Kazemzadeh. Experience with an industry-driven capstone course on game programming. In *Proceedings of the 2005 ACM Technical Symposium on Computer Science Education*, pages 91–95. ACM Press, 2005.

[11] 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.