Challenges and Opportunities in the Design of Game Programming Classes for a Traditional Computer Science Curriculum

Ian Parberry*
Max Kazemzadeh†
Timothy Roden‡
Jeremy R. Nunn
Joseph Scheinberg
Erik Carson
Jason Cole
University of North Texas

March 17, 2010

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 College of Visual Arts and Design. We discuss the design that enables them to provide training for students intending employment in the game industry without sacrificing academic educational depth or the educational needs of mainstream computer science students.

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. 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 more classes and an undergraduate certificate in game programming. After more than a decade and a half 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. Our early alumni are moving upward in the game industry to programming director and producer level positions.

*Communicating Author's address: Dept. of Computer Science & Engineering, Univ. of North Texas, 1155 Union Circle #311366, Denton, Texas 76203–5017, U.S.A. Email: ian@unt.edu.

†Author's Current Address: Gallaudet University, Washburn Arts Center, 800 Florida Avenue, NE, Washington, DC. 20002.

‡Author's Current Address: Department of Computer Science, Angelo State University, San Angelo, Texas 76909-0909.

In sharp contrast to their early history, game programming classes are now gaining acceptance in academia (see, for example, Feldman [7], Moser [11], Adams [1], Faltin [6], Jones [9], Becker [3], Alphonce and Ventura [2], and Sindre, Line, and Valvåg [15]), resulting in a proliferation of new classes and programs nationwide, and a move towards a professionally recommended curriculum in game studies [8]. Unlike 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 that 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 and a half by describing our game programming classes, the design philosophy behind them, and some of the potential pitfalls to be avoided.

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 our introductory class in more detail in Section 4, and our advanced class in Section 5. Section 6 describes two new classes and a recently introduced undergraduaet Certificate in Game Programming at UNT. Section 7 discusses the need for separate lab space for use in game programming classes. The Conclusion contains some data about the impact of game programming on the computer science program at UNT. A preliminary version of the material in this paper appeared in [12, 13, 14].

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 require 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.
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 has the perseverance to see it through to completion.
3. That the applicant is a "team player", somebody who can work in a multidisciplinary team with other programmers and nontechnical people such as artists.
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.

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 [10]:

- Team Attitude: The ability to work in a team to get the job done, without unnecessary friction.
- Self-starters: The ability to work without constant supervision.

- Follow-through: The ability to see tasks through to the end.
- Communication: The ability to communicate with programmers and nonprogrammers.
- Responsibility: The ability to take responsibility for things that have been done, and are to be done in the future.

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 [9]. 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 alone.

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 Department of Computer Science and Engineering 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 the game programming classes. We have in the past attracted a substantial amount of criticism from academics over our choice of DirectX for the graphics API and Visual C++ for the compiler over OpenGL and g++ respectively. In response, the author wishes to make the following observations:

1. The DirectX SDK (Software Developer's Kit) can be downloaded and used free of charge. Visual Studio Express can be downloaded and used free of charge, which with the addition of the Windows Platform SDK (also available for free) and the DirectX SDK can be used for game development under Windows.

2. DirectX is updated every two calendar months. This means that bug fixes are applied quickly. Unlike OpenGL, there is little or no trouble supporting available video cards. A major version of DirectX is released regularly (DirectX 10 will be available within a year), which ensures that the API keeps up with the latest in graphics technology.

3. Students benefit from using in class the same tools and techniques used by a substantial fraction of the game industry.

4. Students should be exposed to as many different compilers and APIs as possible during their academic tenure (this is encouraged in Section 10.2.2 of [4]). Our students are already exposed to open source software including g++ and OpenGL in other Computer Science classes. DirectX and Visual Studio add to this experience, and are in no way intended to supplant it.

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. Not only do they benefit professionally in learning to collaborate across disciplinary lines, but they are strongly motivated to produce better code when they see better art. We will describe more of our collaboration with the College of Visual Arts and Design 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 [15]. Constraints on the type of game being created (as in [1, 2, 3, 6, 7, 9]) 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:

1. 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.
2. 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.

3. 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.
4. Students who are exposed to the internal code of a somple game engine are able to more quickly pick up the details of the proprietary game engine at their first job.
5. 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 Game Programming 1

Our introductory game programming class, now called Game Programming 1, was introduced in 1993 as a special topics class. Despite some initial resistance from faculty curriculum committees, 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.

The intro game programming class 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. It 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

Game Programming 1 is taught in parallel with a game art class taught to art students in the College of Visual Arts and Design at UNT. Part of the art students' grade is to produce the art work for a game programmed by our students. 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.

Final projects 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.

We have used a teaching technique called *incremental development*. Rather than going through the DirectX documentation or tackling a single monolithis game engine, 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).

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

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. 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 is organized into twelve incremental demos as follows:

- Demo 0: Getting started
- Demo 1: Introduction to Direct3D
- Demo 2: Scripting and debugging
- Demo 3: The sprite
- Demo 4: Managing objects
- Demo 5: AI
- Demo 6: The game shell
- Demo 7: Sound
- Demo 8: DirectInput
- Demo 9: The joystick
- Demo 10: Playability
- Demo 11: Persistence

4.1 Getting Started: Demo 0

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.

4.2 Introduction to Direct3D: Demo 1

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.

4.3 Scripting and Debugging: Demo 2

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`.

4.4 The Sprite: Demo 3

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".

4.5 Managing Objects: Demo 4

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.

4.6 Artificial Intelligence: Demo 5

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.

4.7 The Game Shell: Demo 6

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.

4.8 Sound: Demo 7

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.

4.9 DirectInput: Demo 8

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.

4.10 The Joystick: Demo 9

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

4.11 Playability: Demo 10

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.

4.12 Persistence: Demo 11,

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.

5 Game Programming 2

The advanced game programming class, now called Game Programmign 2, 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 a prerequisite.

Game Programming 2 covers interactive real-time 3D animation. The grade for the class is for a 3D game created in groups, a typical group consisting of two programming students and two or more art students from the concurrent art class taught in the College of Visual Arts and Design. An increase in the number of art students per group over Game Programming 1 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 [5]. We have had the greatest success from using a simple game engine which we call SAGE. Students who wish a different experience are permitted to download and use any free game engine.

As with Game Programming 1, 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 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 the College of Visual Arts and Design upgrades its subscription to the 3D modeling tool of choice, leading to a last-minute scramble to provide software tools in time.

To help mitigate these and other problems, we created a simple academic game engine called SAGE. SAGE is designed to provide the minimum requirements for a game, which are:

1. A 3D world

2. that a player can explore in real time,

3. with interesting objects in it,

4. with which the player can interact.

The key adjectives in the preceding list are *real time* and *interactive*. The technology necessary for this includes:

- A graphics renderer, using pixel shaders and HLSL. It is essential for student morale that the rendering engine be reasonably close to cutting edge, and to provide the latest shader technology.
- Objects, including a method for importing 3D models created by artists, and an object manager that takes care of object creation, behaviour, rendering, and destruction.
- A 3D world, consisting of terrain and some method for level-of-detail to increase rendering speed.
- Input from the keyboard, mouse, and joystick to enable the player to interact with the world and the objects in it.
- Collision detection to enable interaction between the player, the objects, and the world.
- A particle engine to enable visual effects that follow from that interaction.

SAGE brings the experience of incremental development to a fully 3D game engine, based on an educational pedagogy that has a proven track record. SAGE includes a sample game, *Ned's Turkey Farm 3D*. The code consists of a sequence of game demos, each showcasing a new feature. The feature is demonstrated in rudimentary form, leaving room for students to enhance it. The trick is getting it complex enough to convey the fundamental principles, yet simple enough for students to understand. SAGE has Doxygen generated documentation, and approximately 200 pages of tutorials. SAGE is developed in C++, uses DirectX 9.0, and is accompanied by Visual Studio project files. It is released under a BSD open source license, and is available on the first author's website and in the Microsoft Developer Network Academic Alliance Curriculum Repository.

SAGE is organized as follows. The following description applies to Demo 6, the complete fully-featured project. The top-level folder contains two subfolders, `Ned3D` containing game-specific code, and `SAGE` containing engine code. The `Ned3D` folder consists mainly of game-specific classes derived from the basic SAGE classes, which we will not describe further here. The `SAGE` folder contains two subfolders, `SAGE Resources` containing resources for the console and effects files for the pixel shaders, and the `Source` folder containing SAGE source code.

`SAGE\Source` contains the following subfolders.

- `Common`: Low-level code, which will be described in more detail below.
- `Console`: The game console.
- `DerivedCameras`: A free camera and a tether camera.
- `DerivedModels`: An animated model using animation frames and linear interpolation, and an articulated model.
- `DirectoryManager`: A directory manager, which manages the organization of resources in subfolders.
- `Game`: The GameBase class, which contains game logic code.
- `Generators`: A name generator and an identifier manager.

- `Graphics`: Graphics related code, including vertex buffers, index buffers, and effects.
- `Input`: Input using DirectInput.
- `Objects`: Game objects and the object manager.
- `Particle`: The particle engine.
- `Resource`: The resource manager.
- `Sound`: The sound manager.
- `Terrain`: The terrain code, including height map and LOD.
- `TinyXML`: TinyXML code.
- `Water`: Code for water animation, including use of the reflection pixel shader.
- `WindowsWrapper`: An abstraction layer for Microsoft Windows specific code.

The `Common` folder is of particular interest, since it contains the low-level code for SAGE. SAGE is based on the freely available low-level code from Dunn and Parberry [5]. `Common` includes the following utilities:

- `AABB3.cpp`, `AABB3.h`: Axially aligned bounding boxes.
- `Bitmap.cpp`, `Bitmap.h`: Bitmap image reader.
- `CommonStuff.h`, `CommonStuff.cpp`: Common stuff that doesnt belong elsewhere.
- `EditTriMesh.cpp`, `EditTriMesh.h`: Editable triangle mesh class.
- `EulerAngles.cpp`, `EulerAngles.h`: Euler angle class.
- `MathUtil.cpp`, `MathUtil.h`: Basic math utilities.
- `Matrix4x3.cpp`, `Matrix4x3.h`: Homogenous transformation matrix code.
- `Model.cpp`, `Model.h`: Simple class for a 3D model.
- `Quaternion.cpp`, `Quaternion.h`: Quaternion class.
- `Renderer.cpp`, `Renderer.h`: Rendering engine (modified somewhat from its original form in [5]).
- `RotationMatrix.cpp`, `RotationMatrix.h`: Rotation matrix class
- `TriMesh.cpp`, `TriMesh.h`: Triangle mesh class.
- `Vector2.h`, `vector3.h`: vector class.
- `WinMain.cpp`, `winmain.h`: Windows dependent code.

The following low-level code was added to Common:

- `camera.cpp`, `camera.h`: Base camera class, from which the free camera and the tether camera are derived.
- `fontcacheentry.cpp`, `fontcacheentry.h`: Encapsulates the Direct3D font class.
- `plane.cpp`, `plane.h`: Math plane class.
- `random.cpp`, `random.h`: Pseudorandom number generator.
- `rectangle.h`: Rectangle class.
- `texturecache.cpp`, `texturecache.h`: Texture cache class.

Phase 1 of SAGE consists of approximately 35,000 lines of C++ code (including header files, code, and comments). The code is distributed into four parts, the Common framework (described above), SAGE code, tinyXML, and code specific to the sample game, *Ned's Turkey Farm 3D*. The number of lines of code in each of these modules is given in Table 1. The code architecture is described in Figure 2, with the foundation being code from Microsoft DirectX and the Windows API, the Common framework being layered on top of that, supporting the SAGE engine, with code specific to the particular game supported by SAGE layered on top of that.

SAGE consists of seven incremental demos, as follows:

Module	Code
Common framework	13,729
SAGE	13,469
tinyXML	4,883
Ned specific	2,750
Total:	34,831

Table 1: Number of lines of code in SAGE.

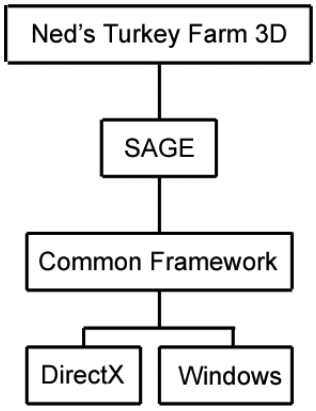

Figure 2: SAGE architecture.

Figure 3: Demo 0 showing the plane model.

- Demo 0: Model importation and display
- Demo 1: Terrain input and rendering
- Demo 2: Shaders using HLSL
- Demo 3: Game engine architecture
- Demo 4: Collision detection
- Demo 5: Particle engine
- Demo 6: 3D sound

5.1 Model Importation and Display: Demo 0

Demo 0 demonstrates the code for reading and displaying a model. The code for Demo 0 shows the programmers how to import a model, render it, and perform simple operations such as rotation and camera motion under user control. In addition, the executable is a useful tool for artists and programmers to check for correct export of models, which can be created using a 3D modeling tool such as Maya or 3D Studio Max (see Figure 3).

Each modeling program has a proprietary file format that changes with each version, the updating of which can cause previously used models to become unusable. Each has facilities for plug-ins to export to a different file format. Some file formats are text, some are binary. Direct3D has a native file format (.X). Other popular file formats exist, eg. Quake II, Quake III models. Managing the input of art assets is one of the biggest startup hurdles in making a game demo. File format converters exist, but our experience with them has in general been less than positive, often resulting in the introduction of degenerate triangles, sliver triangles, missing triangles, detached triangles, and the mangling of origin, axes, normals, and scale.

To help avoid these problems, SAGE uses the S3D format from [5], and includes an S3D plug-in for Maya. S3D is a simple text format that enables the programmer to view the model data directly in a text editor to check for simple errors.

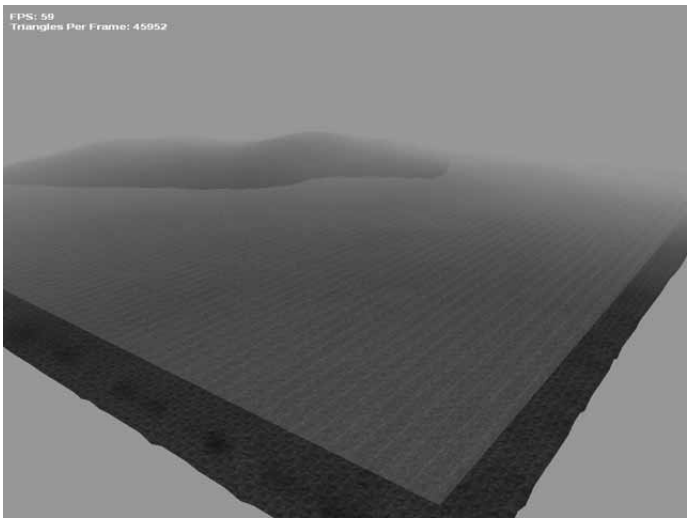

Figure 4: Demo 1 showing ocean and island.

5.2 Terrain Input and Rendering: Demo 1

Demo 1 covers terrain input and rendering. It reads a height map from an image file and renders an island surrounded by a small finite area of ocean (see Figure 4). Simple grid-based level of detail is provided. A free camera can be used to explore the terrain. A simple console allows the user to modify game properties easily.

5.3 Shaders: Demo 2

Demo 2 covers shaders using HLSL. Shaders are provided for texture blending (demonstrated on textures that change with terrain height), and for reflections in water (see Figure 5). A triangle of water that moves with the camera gives the illusion of ocean extending to infinity. We particularly avoided the temptation to create a large number of shaders, preferring to leave that for students. Since shaders are an intricate subject the shader tutorial is the longest of our tutorials, consisting of approximately 50 pages.

5.4 Game Engine Architecture: Demo 3

Demo 3 covers game engine architecture, including objects, an object manager, a tether camera, and DirectInput. Types of objects supported include rigid objects, articulated objects, and animated objects. Articulated objects consist of separate hierarchically organized parts that may be moved or rotated independently, such as the propellor on the airplane and the blades on the windmill in *Ned's Turkey Farm 3D*. Animated objects consist of key frames created by the artist as a set of rigid objects. The SAGE animated object provides in-betweening using linear interpolation. *Ned's Turkey Farm 3D* has crows implemented as animated objects.

5.5 Collision Detection: Demo 4

Demo 4 covers collision detection using axially aligned bounding boxes (AABBs). Collision of objects with terrain, objects with objects, bullets with objects are detected. Object-terrain collision is implemented by

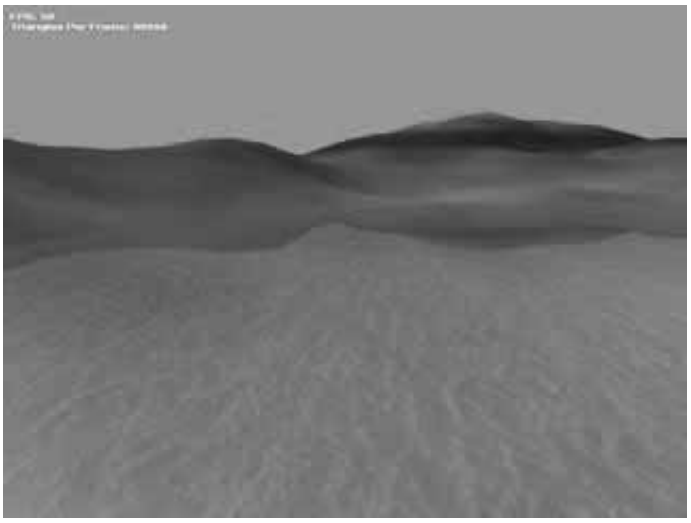

Figure 5: Demo 2 showing terrain reflections and texture blending.

interpolating terrain height within a triangle, object-object collision is implemented using AABB-AABB intersection, and bullet-object collision is implemented using ray-AABB collision detection. In a real game, AABB collision detection would be only the first or primary level of collision detection, designed to quickly eliminate noncolliding objects. Subsequent levels of collision detection, including bounding boxes and bounding spheres at the secondary level, and triangle-triangle collision detection as the tertiary level, are left as possible projects for the student. For educational purposes, SAGE will render AABBs in real time for classroom demonstrations (see Figure 6).

5.6 Particle Engine: Demo 5

Demo 5 covers particle engines and provides a general purpose particle engine that is used in *Ned's Turkey Farm 3D* for explosions, clouds of feathers (see Figure 7), smoke, gunfire flash, and dust raised by a bullet hitting the terrain.

5.7 Sound: Demo 6

Demo 6 covers stereo 3D sound using DirectSound.

6 The Game Programming Certificate

In 2009 we added two more classes in game programming. The first is a class on game math and physics, created in response to student complaints that Game Programming 2 had too much math content. This class was taught for the first time in Fall 2009. Material covered includes:

1. Intro to 3D math
2. Vectors
3. Matrices

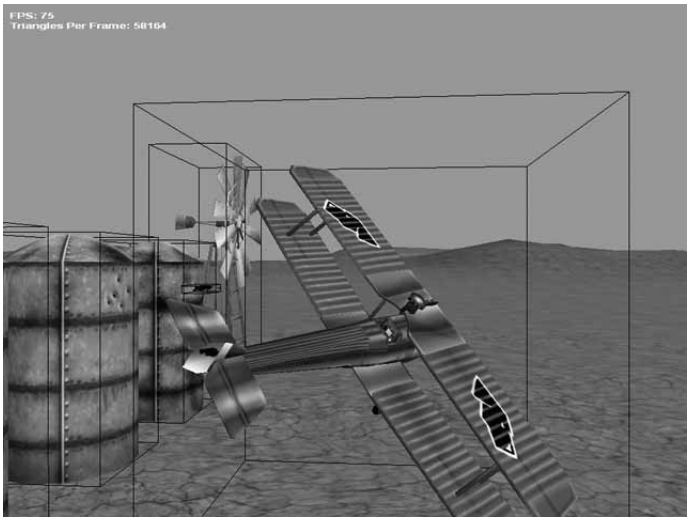

Figure 6: Demo 4 showing AABBs.

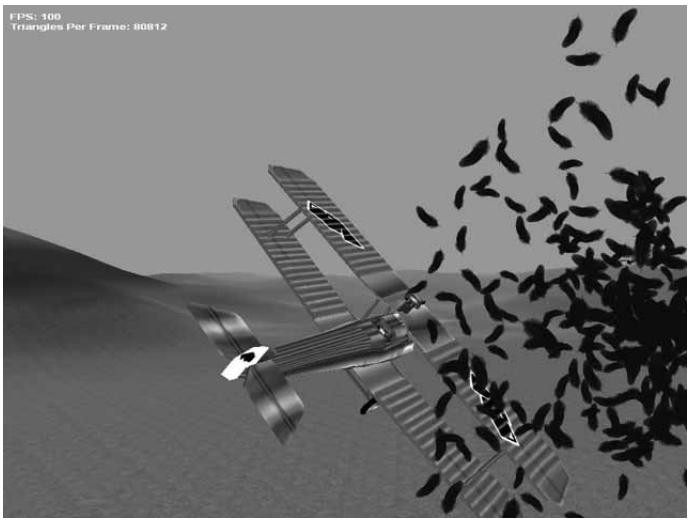

Figure 7: Demo 5 showing cloud of feathers created using particle engine.

4. Intro to game physics
5. Particle motion
6. Newtonian mechanics
7. Multivariate calculus
8. Floating point numbers
9. Ordinary differential equations
10. Quaternions
11. Rigid body mechanics
12. Collision detection
13. Deformable bodies
14. Lagrangian mechanics

The second new class is a Topics class in which the students specialize in the advanced topic of their choice, working in groups. This helps students to fulfill Requirement 4 of Section 2. Topics undertaken recently include water animation, procedural clutter, procedural quest generation, shaders, shadow rendering, and cell processor programming. Undergraduates are encouraged to work with graduate students, which exposes them to academic research and increases the probability that they will return for a graduate degree.

Completion of the four classes, Game Math and Physics, Game Programming 1, Game Programming 2, and Topics in Game Development entitles students to an undergraduate Certificate in Game Programming, starting 2010.

7 The Game Programming Laboratory

Providing a dedicated game programming laboratory proved to be an important requirement when developing the game programming classes. The standard open laboratories provided by universities are unsuitable for game programming classes for several reasons.

- The process of updating software is typically slow and cumbersome, since open labs catered to a wide range of students from various disciplines.
- The hardware, in particular the graphics cards are not up to expected standards.
- Students developing and playing games are distracting to other lab users, and game development students typically run afoul of any rules against game playing, despite the argument that it is required for a class assignment.
- Since game students are required to work in teams with other programmers and artists, a substantial amount of team meetings during development and debugging need to be actually at the keyboard. Open labs are typically 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 and loud.
- 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 approximately 570 square feet, 12 computer workstations, and a file server. The computers are on a special 2-year upgrade cycle, as opposed to the standard 3 or 4-year upgrade cycle, paid for by course fees from the game programming classes. Dual monitors are a must for on-screen debugging.

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 laptop use. The computer workstations are connected to the wired network through a router that serves as a firewall to prevent packet floods generated by rookie game programmers from swamping the building's network.

One corner of the room has been set up with a sofa, a TV, and several game consoles to foster interaction between students and provide an inviting club-like atmosphere. 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 many whiteboards.

The location of the lab is important. Currently it is across the hall from the author's office, which since the office and lab doors are usually open, facilitates the author's interaction with 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. Figures 8 and 9 show the lab layout.

8 Conclusion

We have had success over the last decade and a half 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 College of Visual Arts and Design. 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.

The author believes that 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 10). Figure 11 shows enrollment figures for

Figure 8: Current laboratory layout.

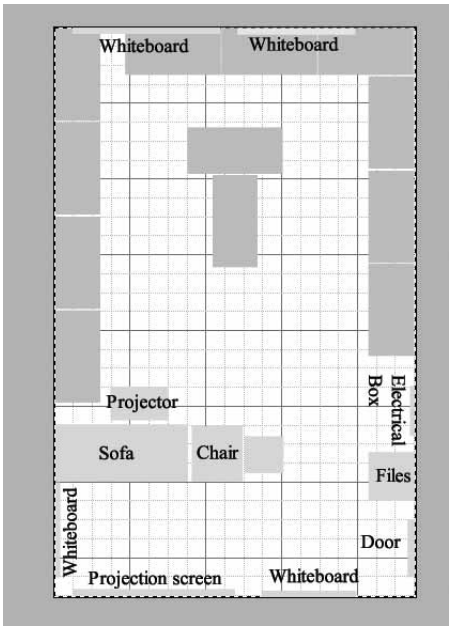

Figure 9: Planned laboratory layout. Unlabelled rectangles are tables.

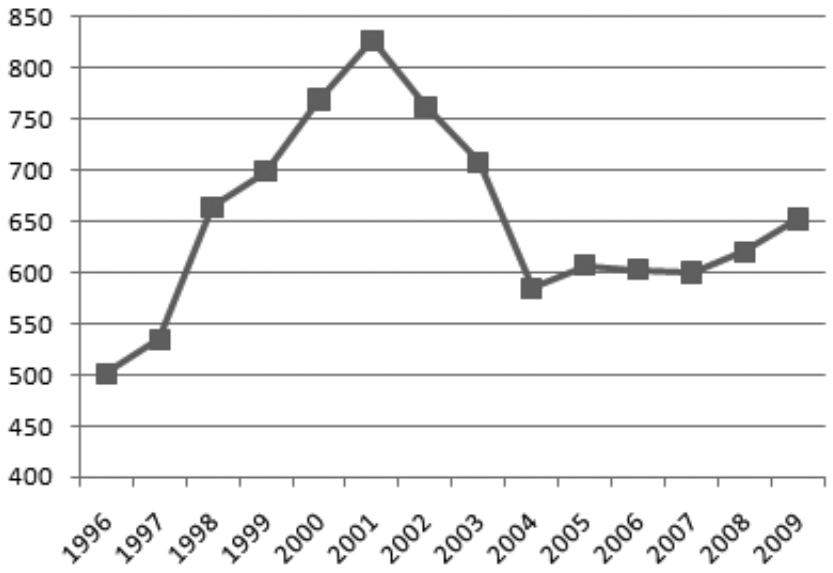

Figure 10: Total enrollments in undergraduate degrees in the Department of Computer Science and Engineering, 1996–2009.

the introductory and advanced game programming classes from to present 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 of graduates have taken game programming.

SAGE can be downloaded from `http://larc.csci.unt.edu/sage`, and was funded by a grant from Microsoft Research.

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

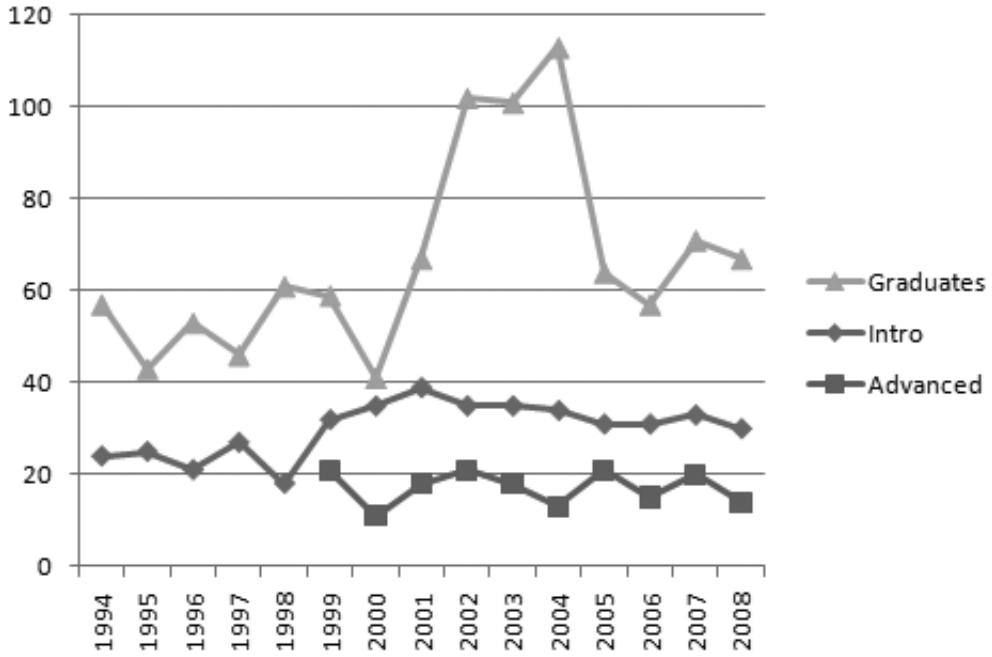

Figure 11: Game class enrollment versus number of CSE Bachelor's degrees awarded by academic year (for example, 1993 means Fall 1993 to Summer 1994).

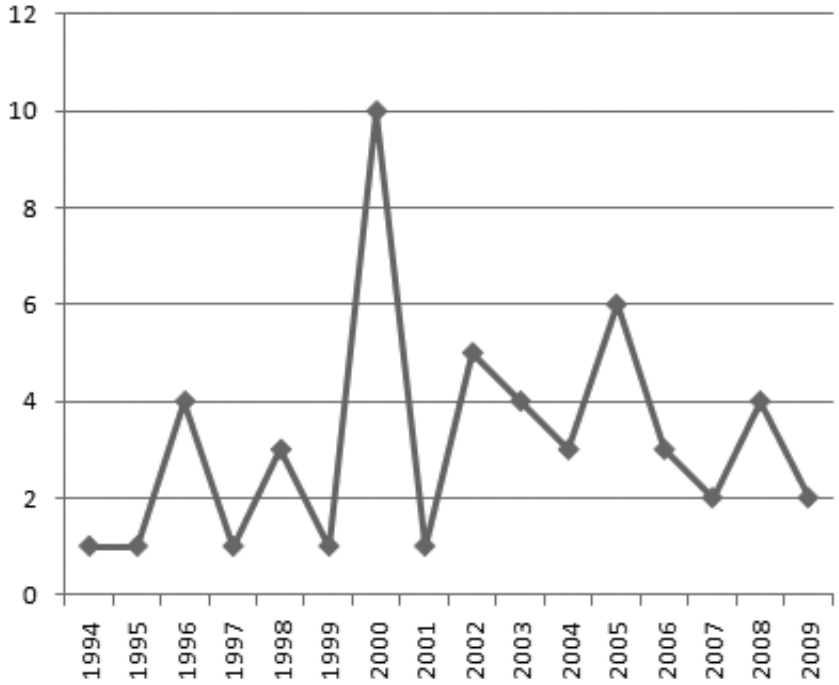

Figure 12: Number of alumni getting jobs in the game industry, 1994–2009.

[6] 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.

[7] 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.

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

[9] 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.

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

[11] 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.

[12] I. Parberry, M. Kazemzadeh, and T. Roden. The art and science of game programming. In *Proceedings of the 2006 ACM Technical Symposium on Computer Science Education*, pages 510–514. ACM Press, 2006.

[13] I. Parberry, J. Nunn, J. Scheinberg, E. Carson, and J. Cole. Sage: A simple academic game engine. In *Proceedings of the Second Annual Microsoft Academic Days on Game Development in Computer Science Education*, pages 90–94, 2007.

[14] 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.

[15] 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.