SAGE: A Simple Academic Game Engine

[Extended Abstract]

Ian Parberry
Jeremiah R. Nunn
Joseph Scheinberg
Erik Carson
Jason Cole
Department of Computer Science & Engineering
University of North Texas
Denton, TX, USA
ian@unt.edu

ABSTRACT

SAGE is a simple academic game engine for use in a game programming class in the undergraduate Computer Science curriculum, designed specifically as a core onto which students can add their own game engine features. SAGE consists of a sequence of demos written in C++ using Microsoft DirectX, each extending its predecessor in a process called *incremental development*. Incremental development is a proven pedagogical technique used for the education of game programmers at the University of North Texas since 1997.

Categories and Subject Descriptors

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

General Terms

Design, Experimentation

Keywords

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. For more information about these classes, see [11, 12].

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 Valvåg [13]), 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.

The students in our game programming classes are usually seniors in the computer science program, who are technologically savvy and experienced programmers. They are usually quite capable of reading the documentation for game APIs, such as Microsoft DirectX, 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. Our game programming classes have a positive effect on undergraduate enrollment in the Department of Computer Science and Engineering at UNT. Out of almost 200 students from the prerequisite classes surveyed in 1993, 49% of students intended to take the introductory game programming class, and 39% said that the class was a contributing factor to their presence in the Computer Science program at UNT (for full figures, see [12]).

Selection of a game engine is a major decision that can make or break a game programming class. Students learning game programming in academia need an engine that is flexible, extensible, stable, and well-documented. Industry game engines such as Vertigo's Quake II .NET, Unreal Technology's Unreal Engine, and Valve's Half Life 2 engine, are large, complex, and relatively complete. An academic game engine should in contrast be small, simple, and incomplete. It should be suitable as a foundation on which students can build, and above all be easy to understand and modify, especially by relatively inexperienced students. It should illus-

trate new concepts in enough detail for students to get started, but should avoid "completeism". It should obey the educational principle "proceed from the known into the unknown".

The main part of this paper is divided into five sections. The first section lists the requirements for a simple game engine, and the technology necessary to implement them. The second section gives an overview of the SAGE project. The third section describes the seven SAGE demos in more detail. The fourth section describes our experience with SAGE in the classroom in Spring 2006. The fifth section discusses our choice of DirectX and Visual C++ for this project.

2. MINIMUM REQUIREMENTS

SAGE is designed to provide the minimum requirements for a game, which are a 3D world that a player can explore in real time, with interesting objects in it, with which the player can interact. The key adjectives in the preceding sentence 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 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.

3. SAGE OVERVIEW

SAGE is a 3D game engine developed as a sequence of demos, each built on its predecessor, in a process called *incremental development*. Incremental development has been used in the construction of a billboard demo in a simple 3D world with limited camera movement (*Ned's Turkey Farm*, see Figure 1) for introductory game programming classes at UNT since 1997 (see [11]). Earlier versions have been published in two books [9, 10]. 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.

SAGE brings this experience 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.

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

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 [3]. `Common` includes the following utilities:

- `AABB3.cpp`, `AABB3.h`: Axially aligned bounding boxes.
- `Bitmap.cpp`, `Bitmap.h`: Bitmap image reader.

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.

- `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 [3]).
- `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.

4. SAGE DEMOS

SAGE consists of seven incremental demos, as follows:

- 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

4.1 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).

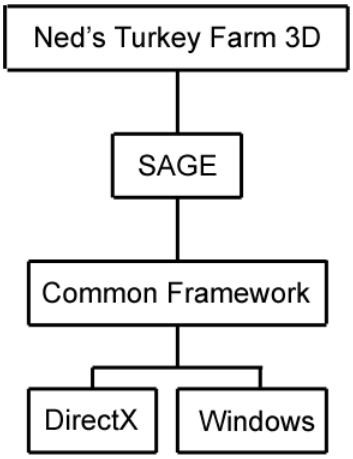

Figure 2: SAGE architecture.

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 [3], 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 3: Demo 0 showing the plane model.

4.2 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.

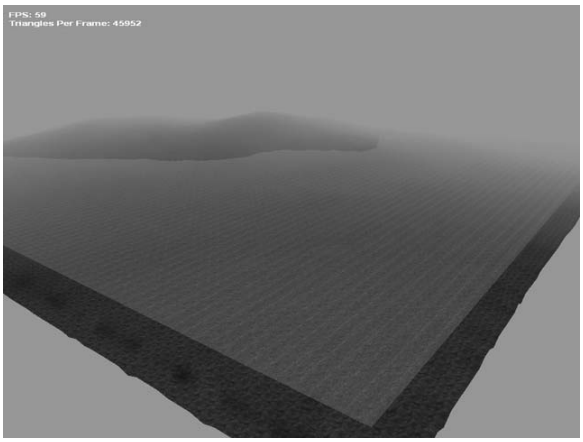

Figure 4: Demo 1 showing ocean and island.

4.3 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.

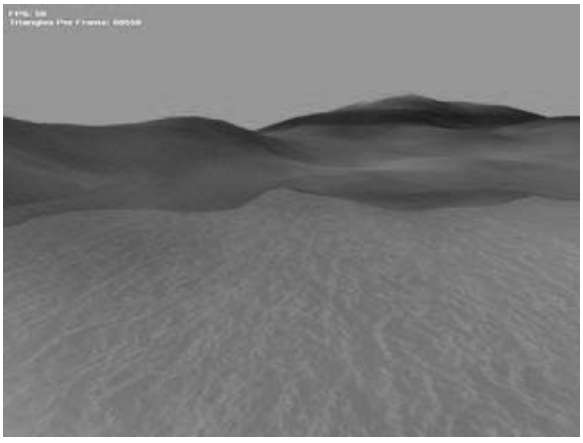

Figure 5: Demo 2 showing terrain reflections and texture blending.

4.4 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.

4.5 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 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).

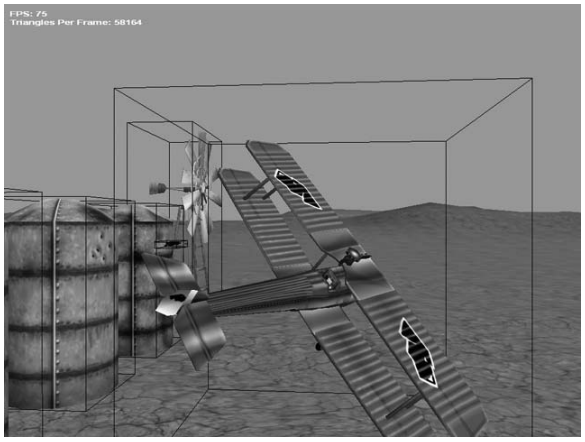

Figure 6: Demo 4 showing AABBs.

4.6 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.

4.7 Demo 6

Demo 6 covers stereo 3D sound using DirectSound.

5. SAGE IN THE CLASSROOM

SAGE was used for the first time in the classroom in Spring 2006 in the first author's CSCE 4220 (Advanced Game Programming) class while the code and tutorials were still under development. The resulting student games were of a higher quality than in previous years, and included the following:

- Duck Hunt: A medieval first-person shooter in which the played floats across a lagoon at twilight in a canoe, shooting flaming arrows at ducks.
- Sink This: A third-person submarine game in which the player attempts to torpedo other submarines.
- Fury Mallard: A third-person shooter in which a duck attempts to kill men in black suits.

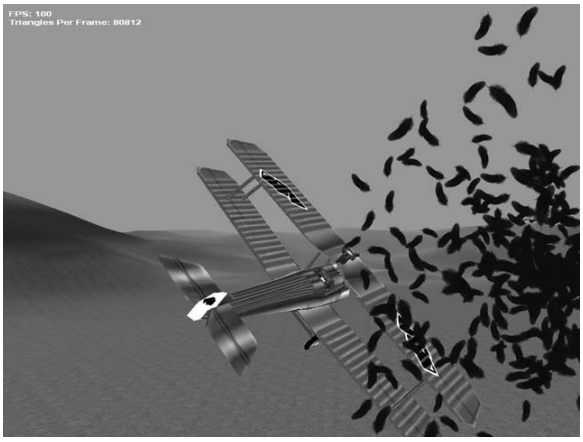

Figure 7: Demo 5 showing cloud of feathers created using particle engine.

- Ghost Hunter: A third-person shooter in which the player attempts to kill zombies.
- Galactic Battlefield: A third-person space shooter.
- Great Space Race: A third person space racing game where the player must navigate between portals against the clock.
- VertiGo: A 3D puzzle game in which the player attempts to navigate a marble through a 3D array of cubes.

6. ON DIRECTX AND VISUAL C++

The authors of this paper have attracted a substantial amount of criticism from academics over their choice of DirectX for the graphics API and Visual C++ for the compiler supported in this project, over OpenGL and g++ respectively. In response, the authors wish 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. We believe that students benefit from using in class the same tools and techniques used by a substantial fraction of the game industry.

4. We strongly believe that students should be exposed to as many different compilers and APIs as possible during their academic tenure. 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.

7. CONCLUSION

SAGE Phase 1 was completed in June 2006, and can be downloaded from `http://larc.csci.unt.edu/sage`. SAGE is funded by a grant from Microsoft Research.

8. 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. 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. *Learn Computer Game Programming with DirectX 7.0*. Wordware Publishing, 2000.

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

[11] 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*. ACM Press, 2006.

[12] 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.

[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.