Computer Science Education
Vol. XX, No. XX, June 2013, 1–18

RESEARCH ARTICLE

Dragonfly – Strengthening Programming Skills by Building a Game Engine from Scratch

Mark Claypool
Computer Science and Interactive Media & Game Development
Worcester Polytechnic Institute, Worcester, MA 01609, USA
email: claypool@cs.wpi.edu
(*Received 00 Month 200x; final version received 00 Month 200x*)

Computer game development has been shown to be an effective hook for motivating students to learn both introductory and advanced computer science topics. While games can be made from scratch, to simplify the programming required game development often uses game engines that handle complicated or frequently used components of the game. These game engines present the opportunity to strengthen programming skills and expose students to a range of fundamental computer science topics. While educational efforts have been effective in using game engines to improve computer science education, there have been no published papers describing and evaluating students building a game engine from scratch as part of their course work. This paper presents the Dragonfly-approach in which students build a fully functional game engine from scratch and make a game using their engine as part of a junior-level course. Details on the programming projects are presented, as well as an evaluation of the results from two offerings that used Dragonfly. Student performance on the projects as well as student assessments demonstrate the efficacy of having students build a game engine from scratch in strengthening their programming skills.

Keywords: game engine, programming, object-oriented design, software engineering, game development

1 Introduction

By the end of their second year, most computer science students have been exposed to a breadth of foundational materials, such as introductory programming, basic data structures and algorithms, and have begun to write programs of moderate size – hundreds of lines of code, perhaps up to even a thousand lines of code. Such students are at the cusp of being able to come into their own as computer programmers. What are often needed to facilitate the next phase in their education are projects that draw upon their knowledge from their earlier courses (e.g. algorithms and object oriented programming) to: 1) reinforce existing programming concepts by application in a new domain, 2) provide an understanding of the inter-connectedness of what may have been mostly disparate computer science topics, and 3) provide a pathway to new knowledge in a domain that inspires and excites them.

Regarding this last point, the need for an inspiration to learn complex programming, computer games provide an opportunity for inspiring projects since games are quite familiar to students in form, if yet still unfamiliar in construction. Today's youth have grown up playing games, with over 90% of kids aged 12-17 playing computer games, most still playing games daily Irvine (2008). Many students choose computer science, or a related educational discipline, because they want to be a game developer as a career when they graduate, or at least would like to develop games as a hobby. Educators can harness this passion, leveraging student interest to engage them in computer programming topics in the context of computer games, thus providing relevance to the materials being taught.

There have been numerous efforts that have sought to leverage students' interest in games in teaching computer science education. These have ranged from using games to teach specific topics, such as recursion Chaffin et al. (2009) to broader topics such as design patterns Gestwicki

ISSN: 0899-3408 print/ISSN 1744-5175 online

DOI: 10.1080/0899340YYxxxxxxxx
http://www.informaworld.com

& Sun (2008) and software architecture Wang (2011). However, while there have been many approaches that have students build games to introduce or reinforce computer science techniques, to the best of our knowledge there have not been systematic efforts for class projects where students build a game engine to enhance programming skills.

A *game engine* is a software system designed to facilitate the development of computer games. As such, a game engine provides services commonly needed by game programmers, including graphics (2d or 3d), input devices (e.g., mouse, keyboard), and event processing (e.g., timers, object collisions). More advanced game engines may include support for networking, physics and artificial intelligence. Game engines are often designed to be *reusable*, allowing game programmers to make many different games relatively easily. Such generality is sometimes supported by allowing game programmers to modify the base game content, such as by providing new models and textures in addition to levels and gameplay rules, giving rise to "mods". Games created with a specific engine are often similar since the game engine is designed with a specific game genre in mind (e.g., a first person shooter game). Game engines are also designed to be *efficient*, tuned to optimize the performance of the most common game operations quickly (e.g., rendering a textured, 3d model on the screen), as well as provide general-purpose functions commonly used (e.g., moving game objects and generating collisions for other objects they hit). Game engines are increasingly useful for applications outside of games, such as scientific research and movie special effects.

Building a game engine provides exposure to and even mastery of many fundamental computer science and software engineering topics: algorithms, software design patterns, and object-orient programming, testing and debugging. In addition, building a game engine affords the opportunity for: 1) programming a sizable software engineering project, a valuable skill students must acquire as the move beyond the "toy" projects that often are done in more introductory courses; 2) in-depth understanding of the workings of a game engine, often leading to more efficient game programmer code that can translate to other, even commercial, game engines; 3) portfolio material demonstrating programming prowess that can be shown to prospective employers during a job search, helping motivate students that may have entered computer science with the intent on being a game developer for a career.

While there are numerous books that provide insights into the components and functionality of a game engine, such as Gregory (2009); Thorn (2010), there are far fewer that examine significant portions of the code of a game engine. The few books that lead a student through actually programming a game engine, such as Yuzwa (2006), tend to provide pages of code that are unexplained and incomplete, or may simply dump the code listing on a CD-ROM, leaving it for the student to decipher. Moreover, learning from a book alone is often unrealistic for everyone but the most talented and disciplined students.

Broadly, education efforts that have developed games Gestwicki & Sun (2008); Wang (2011) have generally not exposed students to game engines, while those that have exposed students to game engines El-Nasr & Smith (2006); Ganovelli & Corsini (2009) have not provided an in-depth understanding of the programming required to do so. The few published education efforts that provide in-depth engine exposure Parberry et al. (2007); Vanhatupa (2011) have not provided the level of understanding nor extensive programming experience that could be provided, in addition to lacking in substantive evaluation of the merits.

Our approach is to construct a series of programming projects that lead students through building a game engine (named *Dragonfly*) from scratch, culminating in their creation of a game using their own engine. In the first project, students complete a tutorial that provides important insights into how to use the Dragonfly game engine from the game programmer's perspective. In the tutorial, students complete a "stock" game, then extend the game to show some mastery in using the engine. In the second project, done in phases, students develop their own version of Dragonfly, first by creating the foundational components of the engine, then building a working, fully functional game engine and, lastly, as time and ability allow, enhancing the engine to be a full-featured game engine.

As detailed in this paper, Dragonfly has been used in two separate course offerings of about 40

total students, providing a basis for evaluation. Objective results indicate students are able to build a fully functional game engine from scratch if used as a major class project assignment, even when the development effort is solo. Subjective results indicate students learn a lot about game engine internals from the course, as well as become significantly stronger C++ programmers. Current efforts are underway to publish a book that details the design of the Dragonfly engine and provides the source code as reference for computer science instructors that wish to use it in their courses.[1]

The rest of this paper is organized as follows: Section 2 lists work related to the efforts presented in this paper; Section 3 presents the Dragonfly approach to teaching students to program a game engine from scratch; Section 4 describes student projects that build the Dragonfly engine and make games with it; Section 5 details our course that uses the Dragonfly approach; Section 6 analyzes results from two offerings of our course; Section 7 discusses some issues related to building a game engine; and Section 8 summarizes our conclusions and presents possible future work.

2 Related Work

Broadly, related work publications fall into three areas – somewhat distantly related efforts that use games in computer science eduction (Section 2.1), more closely related efforts that use game engines in computer science eduction (Section 2.2), and the most closely related efforts that build game engines, or parts of games engines, in computer science education (Section 2.3).

2.1 *Using Games*

There are many related papers that have used games as motivation for teaching computer science.

Chaffin et al. (2009) used a game simulation environment that helped students learn about recursion. Students wrote code that the game environment animated in searching depth-first through a tree. A study using their game with computer science students echoed the enthusiasm students have for games that has been perceived in other work, as well as provided insights into how such games should be constructed.

Volk (2008) presented an effort to teach software engineering in the context of developing a large, commercial-sized computer game. Rather than just extend software engineering into a game, the focus was on a realistic game development setting, including cost and risk assessment. Volk found the gaming-passion of students fueled the project efforts well, with only some supplementary introductory game design material being needed. A sound base in programming skills was required for the class to be a success.

Gestwicki & Sun (2008) used computer game development as a catalyst for teaching object-orientation programming and design pattern integration. Games provided the motivation for students as well as the context for a pattern-rich application. Their paper described six main design patterns taught in the context of an arcade-style, Java-based game. They also presented their teaching methodology, assessment techniques, and results, albeit without quantifiable outcomes.

Wang (2011) described a software architecture course whereby students constructed a game using XNA and C#. Extensive evaluation was made comparing the outcome with an alternative choice students had for the projects – to build a robot-simulation in Java. Evaluation studied the choice of domain compared to the project popularity, complexity of the software architectures produced, student effort and grades earned. Wang found the game development projects could successfully be used to teach software architecture, and that the games projects provided more motivation for the students than the robot simulations.

[1] See `http://dragonfly.wpi.edu/book/`

Kurkovsky (2009) explored whether or not mobile game development provided motivation for learning computer science. Rather than applying the concept in practice, Kurkovsky employed a large student survey to ascertain interest. His work also included a case study showing how mobile game development can be used as a broad introduction to topics in introductory computer science.

The above approaches all have students working with games at the game programmer level, not delving into the functionality of a game engine at all. In contrast, our approach uses the same motivation (i.e., computer games), but provides a much deeper understanding of game functionality through having students develop a general-purpose game engine, and then develop an original game using that engine.

2.2 *Using Game Engines*

There are other papers that use game engines to facilitate computer science education.

El-Nasr & Smith (2006) described how modifying (often called *modding*) existing games using their engines can be used to teach computer science (and other disciplines). They presented two case studies of game-modding in courses that illustrated skills learned by students, as well as increased student motivation. They also described how modding different engines can be used to emphasize different skills.

Claypool & Claypool (2005) described a set of project-based modules to help teach software engineering through designing and creating games. Their approach showed improved class participation and performance and exposed students to different aspects of computer games. Students had choices of game engines to use, such as GameMaker[1] or the Golden T.[2] Subjective assessment from students suggested increased motivation from using the modules.

Ganovelli & Corsini (2009) presented a game engine that allowed students to develop a multiplayer car racing game to stimulate interest and learning in introductory computer science topics. Students implemented the rendering of the scene, while the game engine handled the communication, synchronization and all other aspects of the game. The innovative aspect of their approach was to allow all on-line users to see each other's work, letting students learn from others and motivating them to improve their own.

Muratet et al. (2009) presented a study whereby students developed a serious game as motivation for a class that enhanced programming skills. The game students developed was a real-time strategy (RTS) game built with the Spring RTS engine.[3] As future work, the paper described a teaching experiment that could be used to ascertain if development of a serious game can be adapted to learn programming. Unfortunately, the results of the experiment were not yet known.

2.3 *Building Game Engines*

There are even fewer published papers that describe building game engines as part of a computer science education.

Coleman et al. (2005) presented Gedi, an open source game engine for teaching computer game design and programming in C++. The game engine was used as part of an undergraduate computer science concentration in game development. The authors described the engine in detail, with links to the source code online since the intent is for the engine to be extensible/modifiable by the students. However, there was no evaluation of how using Gedi in courses was, or was not, effective. The engine itself was based on an earlier, out of date effort called Mirus de Sousa (2002). Unfortunately, neither Mirus nor Gedi is currently maintained, and Gedi is several generations old in terms of operating system and software compatibility.

---

[1] `http://www.yoyogames.com/gamemaker/studio/`
[2] `http://www.goldenstudios.or.id/products/GTGE/`
[3] `http://spring.clan-sy.com/`

Parberry et al. (2007) presented SAGE, a game engine designed to be used in computer science undergraduate courses. Specifically, SAGE was designed to let students add their own features to the game engine source code. Some details of SAGE use in the classroom was provided, but evaluation was only analyzed through subjective highlights of the games students produced using SAGE.

Vanhatupa (2011) described efforts and experiences using CAGE, an open source game engine built in C++ and Lua scripts. The graphics for CAGE used Crystal Space[1], a cross-platform development kit for real-time, 3d graphics. Students could modify the C++ code or extend the engine via Lua scripts. CAGE had been used in game programming courses, and the authors highlighted some subjective feedback from students in those courses.

The above papers highlight work that exposes students to the internals of academic, but functional, game engines, allowing students to study and even extend a working engine. However, unlike in our approach, there is no systematic effort to have students build the engines from scratch. Our work has students program a fully functional game engine from scratch, and also unlike earlier work, the paper at hand provides some evaluation of the merits of the approach.

3 Dragonfly

This section first provides the requirements for an approach which has students build a complete game engine as part of a computer science course (Section 3.1), then highlights the areas of programming focus that such an approach provides (Section 3.2).

3.1 *Requirements*

As stated in Section 1, the goal is for students to build a fully functional game engine from scratch. While there may still be value in students getting part-way done with an engine, a partial effort leaves students feeling dissatisfied and, more importantly, unable to use their engine to make games. Also, it must be kept in mind that a game engine, even a small one, is a large programming effort, typically thousands of lines of code. Most computer science students at the start of their junior year have not done class projects of such magnitude, typically only writing "toy" programs of hundreds of lines of code at the most.

Solution: Break the development of the game engine up into three manageable sized phases, where completing all phases results in a fully functional, full-featured game engine. Course lectures provide the design and design rationale for Dragonfly, while the bulk of the students' work outside of lecture time is implementing the design.

An object-oriented language is needed for development, both because of its importance for many other computer science courses, but also because most commercial game development is done in an object-oriented language. Students must have some proficiency with the implementation language chosen as learning a new language at the same time as learning to build a game engine is likely an insurmountable challenge.

Solution: Use C++ for both game development and game engine development. In our curriculum, as in many computer science curricula, the options for an object-oriented language students have exposure to from introductory courses means either C++ or Java. Students had done considerable programming with both languages, generally professing equal comfort with each. However, most game programming and definitely nearly all game engines and engine-type development (e.g., database management systems and operating systems) is done in C/C++.

Visual output is critical for most games, yet a real-time graphics engine is a significant development effort. Even some of the most advanced computer science students are not able to build real-time graphics engines as part of their undergraduate education. The amount of code

[1] `http://www.crystalspace3d.org/`

required for graphics support is typically thousands of lines by itself and could easily dwarf the rest of the engine. In addition, animations and collisions regarding objects should be easy to understand to keep the code manageable. Lastly, our previous experience with programming courses and games found students spending too much time making digital art (so that their games looked good) rather than concentrating on the programming, the focus of the course.

Solution: Use 2d, text-based graphics, specifically *curses*.[1] Curses allows programmers to write text-based applications in a terminal-independent manner. Text graphics (using curses), only requires about 350 lines of engine code. In addition, since only text-based graphics are allowed, students are far less inclined to spend time on art when making a game. Games all end up with a rather "retro" look and feel, which suits many hacker-types quite well. The minimalist art support is actually useful for game design,[2] allowing students to concentrate on the game itself and not how it looks. Having the engine only support 2d graphics greatly simplifies animations and collision detection, too, as object surfaces abstract fairly easily for developing and testing. As a side-benefit, curses also provides for non-blocking input from the mouse and keyboard.

Beyond developing the engine, the goal is for students to build games using the engine. While the engine itself is related to the game, the "fun factor" motivating students stems from game development. Moreover, developing games from the game programmer perspective solidifies the programming knowledge and skills gained from building the engine.

Solution: Have two projects from the game programmer perspective. The first exposes students to the Dragonfly engine, and the second allows students to make a game of their own creation using their engine.

3.2 *Focus Areas*

In building Dragonfly, students focus on the core aspects of a game engine:

- The Game loop: real-time control of game actions
- Input: non-blocking keyboard and mouse
- Objects: Game objects in the world and display objects to the player
- Events: In-game as well as user-defined events
- Physics: Object collisions and velocity for movement
- Display: Drawing and animating sprites

Programming skills are strengthened in the following areas:

- Software patterns: the command pattern (encapsulating events), the iterator pattern (traversing containers), the observer pattern (objects indicating interest in events), and the singleton pattern (for game engine managers).
- Inheritance: base objects to derived game objects and display objects to derived game code objects (e.g., Hero). Includes run-time polymorphism, and dynamic and static type casting.
- Real-time: game loop programming and events dependent upon precise timing (milliseconds of granularity).
- Data structures: containers for lists of objects along with iterators, and scene graph implementations, with quad trees an option.
- Input/Output: reading sprite resources from a file, output log messages to a file, and non-blocking keyboard and mouse input.
- Debugging: debugging from game code through libraries (the engine), debugging pointers and memory allocation, and exposure and refinement to using a debugger (`gdb` and `ddd` in our case).
- Testing: unit testing and functional testing.

[1] Actually, *ncurses*, the most widely known, open source implementation of curses.
[2] `http://www.perlenspiel.org/`

- : Performance: tuning objects and collision representation via a scene graph and measuring performance with benchmarks and profiling.

In all, a typical student implementation of the Dragonfly engine is about 5500 lines of code, spread over 30 classes. For reference, the Dragonfly class documentation[1] can be found online at `http://www.cs.wpi.edu/~claypool/papers/dragonfly-projects/include/`.

4 Projects

This section describes the projects students work through in the Dragonfly approach. The individual projects can be found online at `http://www.cs.wpi.edu/~claypool/papers/dragonfly-projects/`. Point breakdowns and grading rubrics are also provided for each project.

4.1 *Project 1 – Catch a Dragonfly*

The first project is for students to get used to Dragonfly, typically their first exposure to a text-based, 2d game engine. Students work through a tutorial that has them make a simple, "stock" game using Dragonfly. This helps students better understand a game engine by developing a game from a game programmer's perspective, providing the foundational knowledge needed for building their own Dragonfly game engine in project 2 (Section 4.2, and designing and developing their own game from scratch with it in project 3 (Section 4.3).

In project 1, students:

(1) Visit the Dragonfly Web page and briefly familiarize themselves with the contents. The Web page includes a download of the Dragonfly game engine (compiled – no source code), documentation with details on the classes and methods for the game programmer, and links to games and utilities that may be helpful for subsequent projects.
(2) Download the Dragonfly game engine for the environment of choice (both Linux and Cygwin[2] on Windows are supported) and setup their development environment. This means ensuring all the needed external libraries are in place (e.g., curses), installing the Dragonfly libraries and header files in an appropriate place[3], and creating a basic Makefile and simple test program to be sure development can proceed. The same basic setup is used for the students' own game engine development in project 2.
(3) Complete the tutorial, available from the Dragonfly Web page.[4] The tutorial has students build an arcade-style shooting game, called *Saucer Shoot*, where the player flies a space ship into combat against an ever-increasing number of enemy saucers. The tutorial has all sprites needed for development as well as working sample code for students to reference. Figure 1 shows screen shots from the Saucer Shoot game students build. The splash screen is shown when the game starts, followed by the opening where players can select 'p' to play. Once playing, the player controls the spaceship on the left of the screen until hit by a saucer, resulting in the game over screen.
(4) Extend the Saucer Shoot game in a meaningful way by adding 10% or more functionality. For example, student's may add additional weapon types or enemies, health and/or multiple lives, a high score table, or something entirely of their own creation. The actual 10% extension done is up to each student, but s/he indicates what is done with brief documentation when submitting the assignment.

Students work alone for project 1. When done, students turn in a source code package with all

[1] Generated by Doyxgen
[2] Cygwin provides a Unix-like environment for Windows. Cygwin is free under the GNU GPL. See: `http://www.cygwin.com/`
[3] Dragonfly can be installed in user space – root/administrator permissions are not needed.
[4] `http://dragonfly.wpi.edu/tutorial/`

Figure 1. Screenshots from the Dragonfly tutorial game, *Saucer Shoot*. Top left: splash screen, top right: opening, bottom left: game play, bottom right: game over.

code necessary to build their games, including header files and any needed additional sprites (depending upon their extension). In addition, each student includes a Makefile for compiling their game, a README file explaining the platform, files, code structure, and anything else needed to understand (and grade) their game, and a GAME file providing a short description of the additional 10% functionality extension to the Saucer Shoot tutorial game, including indicating the code written.

4.2 *Project 2 – Dragonfly*

In the second project, students build their own version of Dragonfly. Project 2 is broken into three parts: A) *Dragonfly Egg* (Section 4.2.1), B) *Dragonfly Naiad*[1] (Section 4.2.2) and C) *Dragonfly* (Section 4.2.3), that build upon each other to end with a fully functional, full-featured game engine. Since it is critical that game engine code be easily understood (from the game programmer's perspective) and, equally importantly, robust, the three projects are structured such that completing parts A and B provides for a fully functional, if somewhat limited, game engine. This level of proficiency enables students to proceed to project 3, where they make a game using their engine. Completing part 2C provides for a full featured game engine, with functionality that makes it easier to create a broader range of games.

For timing and grading, the due dates are staggered so that most of the time is allocated for part A and part B, but there is still time for completing part C for the top students in the class. In addition, points are allocated such that completing part B is sufficient for earning a "B" grade for project 2, while fully completing part C provides an opportunity for earning an "A" grade for project 2.

Students are informed that it is much better have tested, trusted robust code that only implements part A and part B then it is to have buggy, partially working code that attempts to get into part C. Since students make use of their own engine for project 3, most tend to heed this advice.

Students work alone for all of project 2. While group work is important for many aspects of software engineering, including game development, developing the engine solo ensures students

[1] A *naiad* is the name for the aquatic larva of a dragonfly.

have complete and deep understanding of both the game engine and the programming skills needed to develop it – there is no way to "hide" behind a more experienced teammate. That is not to say students are alone, however – discussing the project with other students is encouraged, even for help in debugging each other's code. The line is drawn at not allowing sharing of code in that each student must write all the engine code him/herself.

All development is done in C++. Students are expected to be familiar with C++ from earlier computer science classes, but are not expected to be experts in the language. While development is done as "homework" outside of class, the requirements and design of Dragonfly are presented in class, with discussions of design rationale, implementation choices and alternatives, and more advanced features.

Individual classes, with high-level descriptions of attributes and methods, are provided in the project writeup at `http://www.cs.wpi.edu/~claypool/papers/dragonfly-projects/proj2/`.

4.2.1 Dragonfly Egg

Part A of the project is to construct the foundations of a game engine that provides the following capabilities:

- Game initialization: Start and stop gracefully.
- Logging: Write time-stamped messages to a file.
- Object support: Add and remove game objects. Objects support 2d game world positions for objects.
- Game loop: Run a game loop with: 1) A fixed update rate (e.g. 30 Hz), and 2) updates sent to all objects each loop

To implement this functionality, students develop, code and test about a dozen base classes.

No visual depiction of the game is required for part A. Instead, all output is done via printing to the screen or to a log file via the logfile manager functionality built into the game engine. As suggested above, at the successful completion of part A, students do *not* have a game engine. Instead, they have a robust, foundational code base they can build upon to get a functional game engine in part B.

4.2.2 Dragonfly Naiad

Part B is to continue construction of the game engine, each student using their own code base from part A, adding the following additional capabilities:

- Output: Support 2d, text characters with color. Provide a clean refresh each game loop.
- Input: Accept non-blocking keyboard and mouse input. Send input to interested game objects.
- Collisions: Provide a "solid" attribute for game objects. Detect collisions between solid objects. Send an event to both objects involved in a collision.
- Misc: Provide deferred, batch removal of game objects. Support event filtering, where objects register when interested. Provide support for an "altitude" attribute for game objects to support layered drawing.

All of the above capabilities must be thoroughly tested, bug-free and ready for a game programmer to make a game (the students themselves, in project 3).

4.2.3 Dragonfly

Part C is to continue construction of the game engine, each student using their own code base from part A and part B, adding the following additional capabilities:

- Sprites: Provide multi-character frames. Associate one or more frames with a game object. Play frames in sequence to achieve animation. Support "slowdown" of animation to less than one frame per game loop.
- Resource Management: Read sprite data from files. Provide bounding boxes for game ob-

jects. Allow game objects to be larger than a single character (for movement and collisions). Associate bounding boxes with sprites.
- Camera Control: Allow the game world to be larger than the screen, providing a "viewport". Enable free viewport movement around the game world, including the ability to follow one object (e.g., the player's avatar).
- Object Control: Support "velocity" for game objects with automatic updates.
- View Objects: provide an alternative (to game objects) object that supports "heads-up display" functionality.
- Misc: Gracefully handle a "kill" signal, as well as random number seeding.

As for project 2B, all of the above capabilities must be thoroughly tested, bug-free and ready for a game programmer to make a game (the students themselves, in project 3).

For each part, students turn in a package with all code necessary to build their game engine, including header files and a Makefile for building their engine. Game programmer code (i.e., code someone would write using their engine) is required to demonstrates the full functionality of what they have built (so far). This can be more than one program, if needed. Documentation is required to explain the platform, files, code structure, how to compile their engine and game code, and anything else needed to understand (and grade) their game engine.

4.3 *Project 3 – Dragonfly Spawn*

In project 3, students use the Dragonfly game engine they built in project 2 to make their own, original game from scratch. The end result is expected to be a robust (bug-free), playable, and balanced game (it may even be fun).

Like a typical large game development effort, the project is broken into several milestones: plan, alpha and final. Each milestone is submitted and graded separately, while all apply towards the total project 3 grade. The intent of the milestones is to provide production guidance to yield a fully-functional, complete, playable game built with their own game engine.

Students work in teams of two for project 3. Students are free to partition the work among the team as they see fit, but all team members are encouraged to help (say, with design and debugging) and be knowledgeable (in terms of how the game code executes) for all parts of the game.

Development must be in C++ using their game engine from project 2. Under exceptional circumstances (e.g., both partners not completing project 2b), students are allowed to use the pre-made Dragonfly engine from project 1. No engine source code is provided, however, only the pre-compiled engine.

4.3.1 Plan

Student teams provide a game plan within the first two weeks of the project. The plan document provides a detailed description of the game they plan to build, including the technical challenges it entails, a bit about any significant artistic aspects of the game, and the timeline to successfully complete development in the time provided. In planning, students are asked to draw upon experiences from other classes (e.g., other programming courses), to inform the creation of the plan document. While the actual length of the plan is not a requirement, as a guideline the plan is expected to be approximately 2-3 pages – much less and students have probably have not supplied enough details.

For the plan submission, students turn in a written document.

4.3.2 Alpha

At alpha stage, the student games have all of the required features implemented, but not necessarily working completely correctly. Game code must be tested thoroughly enough to eliminate any critical gameplay flaws, but minor bugs or glitches may be present.

Games must compile cleanly and be runnable, even if all aspects of gameplay are not available

from one program. Separate features of the game may be demonstrable from separate game code programs (e.g., separate game programs illustrating a kind of weapon or a specific opponent).

Games are likely not yet be finally balanced nor the levels designed for all experiences (beginning to advanced) of game player.

Games may contain some placeholder art assets. For example, in the alpha release, a simple, non-animated square may be used for an opponent with the intent of creating a figure and frames of animation for the final version.

For the alpha submission, students hand in a package with all the source code necessary to build their game engines and their games. All header files must be included, as well as Makefiles for building the games.

4.3.3 Final

The final version of all games has all game content complete – design, code and art. Games must be tested thoroughly for bugs, both major and minor, removing all visual and gameplay glitches. Game code must compile cleanly and be easily runnable. Upon startup, instructions for the player on how to play must be readily available, and with clear indications on how to begin play. Gameplay must be balanced, providing appropriate difficulty for beginners and/or early gameplay, with increased difficulty as the game progresses. Games must have a clear ending condition (i.e., winning or losing) and the player must be able to exit the game easily and cleanly.

For the final submission, students submit their engine and game, with necessary support files and Makefiles. The typical READMEs are required, as well as DESIGN documents providing all the details in the plan, but updated to reflect the games as actually built. For example, the functionality, milestones and work responsibilities need to be updated from the plan to reflect the development. Major deviations from the original plan must be be noted.

5 The Dragonfly Approach

A course designed around the Dragonfly approach, building a game engine from scratch using the projects detailed in Section 4, has been incorporated into the yearly course offerings at our University. The educational objectives of the course are:

(1) Understand the structure and design of a game engine
(2) Demonstrate understanding of a game engine from a game programmer's perspective by extending a simple game
(3) Use a game engine to create a complete, original game from scratch
(4) Use iterative design and development practices to create a playable game
(5) Understand how software engineering techniques can be applied to creating the parts of a game engine
(6) Gain experience and develop skills in working in a team on a software project of significant size, with a short deadline

The course is taken by juniors, with students typically having about four computer science programming courses before this course. These background courses provide an introduction to programming, object-oriented design concepts, system programming concepts, algorithms and data structures. Students have used C++ in at least two of the courses, but about half the students claim to be more comfortable with Java than with C++. A small set of the students (about 1/3) have already had software engineering, another junior level course.

The in-class material begins with about a week of introductory material on game engines and specific C++ concepts needed (somewhat as a review, but some of which is new), but proceeds quickly to concepts specific to the design and development of Dragonfly.

The first 2/3rds of the course has in-class material specific to the Dragonfly engine, with generalities made to other engines where appropriate. The last 1/3rd of the course focuses on other technical topics that are not incorporated into the student's game engines. These include

iterative development (which is helpful for students as they develop games using their engines as described in Section 4.3), performance tuning (benchmarking, profiling and code optimizations) and pathfinding (including A*).

For projects, the first project (creating the tutorial game and extending it, Section 4.1) takes about 1/6th of the course. The second project (building the Dragonfly game engine, Section 4.2) takes a bit over 1/2 of the course. The third project (creating a game using their engine, Section 4.3) takes about 1/3rd of the course.

There are two short, in-class exams. The exams are designed to evaluate whether or not students learned some of the high-level concepts taught in the course that may not have been adequately tested in the programming projects because: 1) students are able to get their code working if they persist enough and with enough help, and 2) programming alone does not adequately reflect understanding of the concept.

The breakdown for overall course grading is depicted in Table 1. The projects make up the bulk of the course grade, with exams an appropriately small fraction. Still, the exam grades comprise approximately an overall letter grade. The projects themselves are dominated by building the engine, but understanding the engine from the game programmer's perspective (project 1) and making a game from scratch (project 3) are needed to pass the course. Group work, done only for project 3, is small relative to the amount of solo work, making it nearly impossible for weaker students to "hide" if their programming skills or effort are not sufficient.

Table 1. Grade breakdown for Dragonfly course

Item	Name	Percent
Project 1	Catch a Dragonfly	10%
Project 2	Dragonfly (Egg, Naiad, Dragonfly)	50%
Project 3	Dragonfly Spawn (Plan, Alpha, Final)	25%
Exams	Mid-term and Final	15%

6 Results

This section presents the results from using Dragonfly projects from Section 4 in the course described in Section 5. Subsections include: an overview of the offerings (Section 6.1), analysis of the grades (Section 6.2), analysis of the degree to which the course objectives were met (Section 6.3), and examination of the student responses to the course and projects (Section 6.4).

6.1 *Overview*

The Dragonfly course has been taught twice, once in Spring 2012 and once in Fall 2012. Both offerings were identical in terms of project assignments and very similar in terms of in-class lectures. Enrollment was 14 for the first offering and 24 for the second offering. The course consisted primarily of juniors, as shown in Table 2, with demographics of each offering following those of the university as a whole, predominantly male (91%) and white (68%).

Table 2. Enrollment for Dragonfly courses

Offering	Fresh	Soph	Jnrs	Snrs	Ttl
Spring	0	2	12	0	14
Fall	0	3	17	4	24
Percent Total	0%	13%	76%	11%	100%

Table 3. Grades for Dragonfly courses

Offering	A's	B's	C's	NR's
Spring	6	5	0	3
Fall	9	8	5	2
Percent Total	39%	34%	13%	13%

6.2 *Grades*

Table 3 lists the grades students earned for the two offerings, broken down by number of each letter grade in the columns with a total and overall percentage in the bottom row. The 'NR'

field stands for "no record", which indicates a student did not earn a C or higher for the course.[1]

Generally, those that earned passing grades achieved some measure of success in their game engine and game. Those that earned B's had a working game engine (through project 2B), but typically few, if any, of the features in the full-featured game engine. Those that earned A's completed much, if not all, of the full-featured game engine and created a successful game with it. The high percentage of A's and B's suggests most students did well in the bulk of the course.

Given the relative similarity in overall grades for the two offerings (more A's than any other single grade and A's and B's dominant), for all subsequent grade analysis the scores for the two offerings are combined.

Table 4 lists a breakdown of the grades for Project 1 (extend a simple game), Project 2B (build a functional game engine), Project 2C (build a full-featured game engine), and Project 3 (build a game from scratch). The 'NC' field indicates "no credit" in that not enough of the project was complete to garner a passing grade.

Table 4. Grades for Dragonfly projects

Assignment	A's	B's	C's	NC's
Project 1	26	7	4	1
Project 2B	22	9	3	4
Project 2C	10	2	6	20
Project 3	18	11	6	3

Table 5. Assessment for Dragonfly projects

Assignment	Completed	Good	Excellent
Project 1	97%	87%	68%
Project 2B	89%	82%	58%
Project 2C	47%	32%	26%
Project 3	92%	76%	47%

Subjective assessment of the project grades provides insights into student accomplishments, shown in Table 5. Projects that earned a C or higher are labeled "completed", projects that earned a B or higher "good", and projects that earned an A "excellent".

From Table 5, most students were able to get a functional game engine, shown by an 89% completion rate for project 2B, with a majority of them (82%) good. There was less success at getting a full-featured game engine, shown by a 47% completion rate for project 2C, but most of those that did complete 2C were good and half of them excellent. The large majority of student were able to build games using the game engine, shown by over 90% of the students completing project 1 and project 3. Most of these projects were good and most of the project 1's were even excellent.

6.3 *Objectives*

The degree to which the course objectives (see Section 5) are met is assessed largely through students' performance on the projects.

Objective 1: *Understand the structure and design of a game engine* and Objective 5: *Understand how software engineering techniques can be applied to creating the parts of a game engine.* The software engineering techniques (e.g., various design patterns) are presented in class materials and then applied by the students in project 2B and project 2C. These objectives are demonstrated by project 2B, where students build a functional game engine. The large majority (89%) of students meet these objectives and meet them well (82% good). Additional demonstration is achieved by about half (47%) the students that completed project 2C (the full-featured game engine).

Objective 2: *Demonstrate understanding of a game engine from a game programmer's perspective by extending a simple game.* This objective is demonstrated by project 1 where students complete a tutorial making a simple game, then extended the game functionality in a manner of their choosing. The overwhelming majority (97%) of students met this objective, with most meeting it quite well (87% good, 68% excellent).

[1] At our University, courses with an NR do not appear on a student's transcript.

Objective 3: *Use a game engine to create a complete, original game from scratch*, Objective 4: *Use iterative design and development practices to create a playable game* and Objective 6: *Gain experience and develop skills in working in a team on a software project of significant size, with a short deadline*. Project 3 has iterative development with the design and development proceeding through different milestones (plan, alpha, and final). These objectives are demonstrated by project 3, where students build a game of their own design using their own game engine (project 2). The overwhelming majority (92%) of students met this objective, while most of them did this well (76% good).

6.4 *Student Responses*

Subjective responses from the students themselves were obtained by anonymous surveys, distributed and filled out during class time at the end of the course.

The first question is "My overall rating of the quality of this course". Responses are provided on a 5-point scale, with 1 labeled "very poor" and 5 labeled "excellent". Other points are unlabeled. Figure 2 depicts the distribution of responses to question 1. The x-axis in the numeric response and the y-axis is the percentage of the total responses. Each bar represents the percentage of respondents that gave the indicated numeric score. The blue asterisk (*) shows the mean. The responses being all positive, with the mode "excellent" and the mean 4.5, suggests the overall, the course has value.

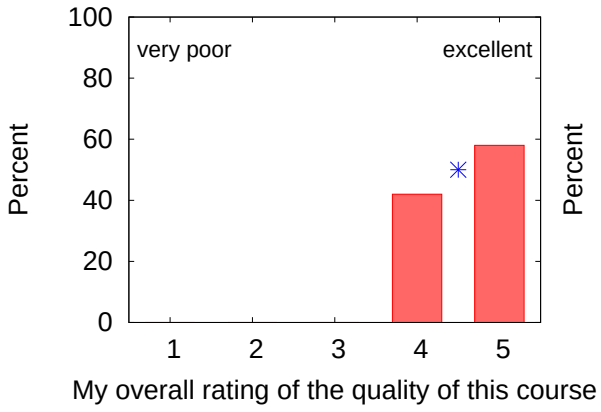

Figure 2.

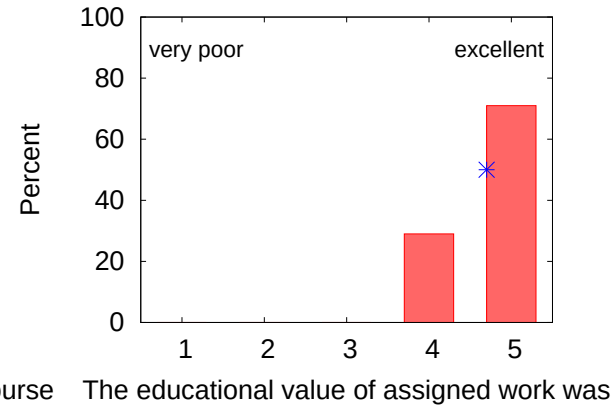

Figure 3.

The second question is "The educational value of the assigned work was", with responses provided on the same 5-point scale as for the first question. Figure 3 depicts the distribution of responses to question 2, with axes and trendlines as for Figure 2. Here, the responses are even more positive, with the mode "excellent" and the mean 4.7, suggests the game-engine focus of the assigned work is of high educational merit.

The third question is "The amount I learned from this course was". Responses are provided on a 5-point scale, with 1 labeled "much less", 5 labeled "much more" and other points unlabeled. Figure 4 depicts the distribution of responses to question 3, with axes and trendlines as for the previous figures. The responses are all positive, with most students near the "much more" end of the scale. The responses here complement the results from questions 1 and question 2 in indicating students learned a lot.

The fourth question is "On average, what were the total hours you spent per week on all activities related to this course." Responses are provided on a 5-point scale, with labels in hourly ranges of: 0-8, 9-12, 13-16, 17-20 and 21+. Figure 5 depicts the distribution of responses to question 3, with axes and trendlines as for the previous figures. Since in-class lectures are only 4 hours a week, with no assigned reading, most of the course time reported is spent on the

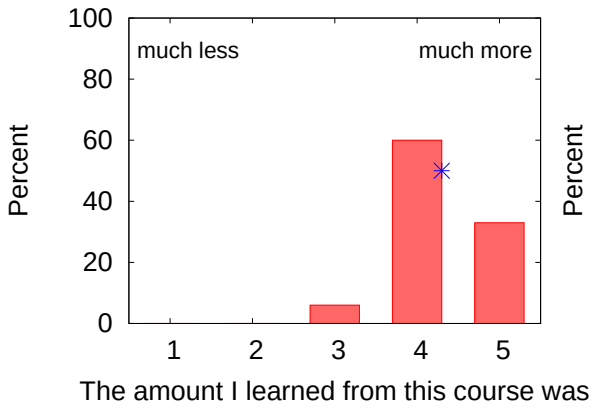

Figure 4.

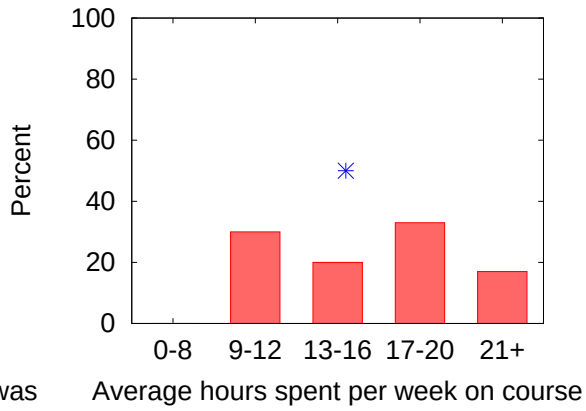

Figure 5.

projects. The mean, 15.9 hours, is computed by taking each response as the middle of each bar, and using 22.5 for the 21+ bar. There is a fairly broad and uniform distribution of hours spent. It should be noted that Worcester Polytechnic Institute (WPI) is unusual in that students take only 3 courses at a time for a 7 week term (there are 2 terms each semester), as opposed to most Universities where students take 5 or 6 courses at a time for a 14 week semester. Since WPI's expected number of hours for students to spend on each class is about 17 hours per week (although most classes fall under this number), the data suggests this course meets the target and is above the university average. For most, the hourly average is manageable for a full-time student.

The fifth question probes student opinions on an alternative game-engine strategy – namely, learning from the source code of a commercial engine (e.g., Quake or C4): "I would rather build my own game engine than have projects based on a commercial game engine." Responses are provided on a 5-point scale, with 1 labeled "strongly disagree", 5 labeled "strongly agree" and other points unlabeled. Figure 6 depicts the distribution of responses to question 5, with axes and trendlines as for the previous figures. The distribution of responses and the mean (4.0) suggest students prefer to build a game engine from scratch rather than try to understand, and then extend or modify, an existing commercial engine.

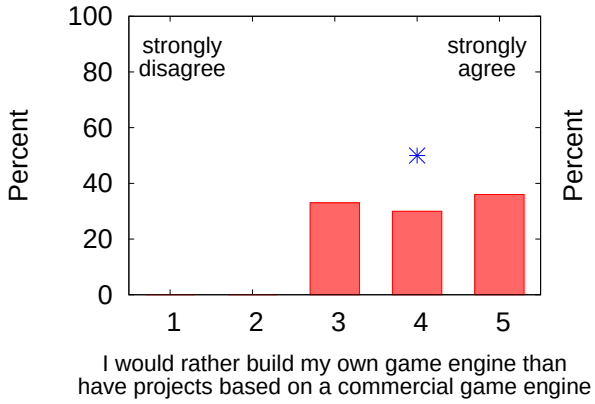

Figure 6.

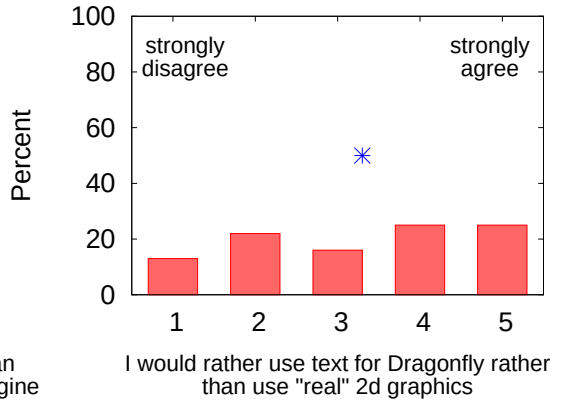

Figure 7.

The sixth question attempts to ascertain attitude students have towards working with text-based graphics, rather than "real" graphics: "I would rather use text for Dragonfly rather than use r̈eal2̈d graphics." Responses are provided on the same 5-point scale as for question 5. Figure 7

depicts the distribution of responses to question 5, with axes and trendlines as for the previous figures. The distribution of responses is quite broad with about an equal number of students being content with the text-based graphics as those that are discontent (the mean is 3.3). This suggests there is not uniform resistance to students in using text-based graphics despite their lack of significant use in commercial games, and considerable support from many students.

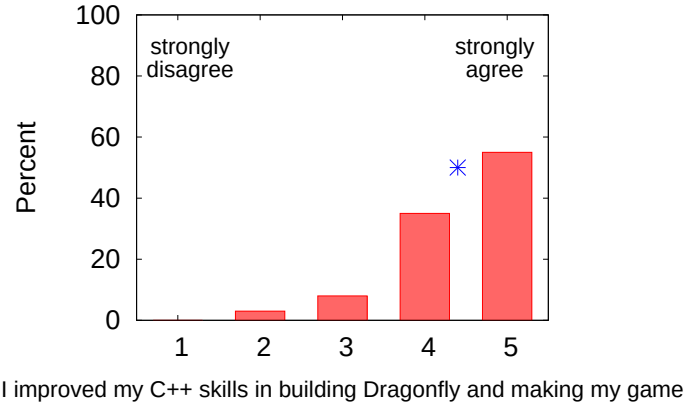

Figure 8.

The seventh question gathers student opinions on how the projects improved their C++ programming skills: "I improved my C++ skills in building Dragonfly and making my game." Responses are provided on the same 5-point scale as for question 6. Figure 8 depicts the distribution of responses to question 6, with axes and trendlines as for the previous figures. The distribution of responses is heavily skewed to the right (the mean is 4.4), with nearly all students believing the projects improved their C++ programming. While this is a subjective opinion, the results are encouraging since strengthening programming was a major goal of the Dragonfly approach.

7 Discussion

While the approach of developing a game engine has merits, there are a few issues worth discussing beyond those described thus far.

If students fall behind in developing the Dragonfly game engine, they may never recover. For example, it is impossible to successfully complete project 2B without having completed 2A, and likewise for 2C without 2B. Although we have not tried it, this shortcoming could be mitigated by providing those students that fail to complete project 2A with a working implementation they could use for 2B. This is similar to what we do allow for Project 3 – namely, if students have not developed their own game engine to sufficient capabilities (or sufficient robustness), they are allowed to use our Dragonfly implementation. Note, however, in this case they do not get the Dragonfly source code but instead get a pre-compiled library and header files they can then use for the game code. However, in the case of 2B, not providing the source code for project 2A (and 2B) is more problematic, since in doing 2B (and 2C), students need to re-factor their code with enhanced functionality.

The code base affords many opportunities for advanced computer science topics, also. For example, Dragonfly can be extended to support various artificial intelligence techniques, such as *pathfinding* or *flocking*. Or, multiplayer game support can be provided in Dragonfly by adding networking support, either client-server or peer-to-peer. These advanced topics can be used independently in appropriate courses, or students may use their own Dragonfly code base for courses that appear in sequence.

The benefits of focusing on a game engine, both for the motivational reasons (i.e., computer games) and for the coding challenges, can be obtained by alternate means. One alternate approach is to use a commercial game engine (e.g., *Unity*) for reference with projects for building a game using that engine. In doing so, students get the benefits of a game engine as motivation, with alternative benefits of developing skills in understanding a complex application programming interface (API). Programming skills can still be strengthened by building games using this engine. Commercial engines that provide source code (e.g., *Unreal Tournament*) provide the additional benefit of exposing students to a large, complex code base. In earlier offerings, we have used Terathon Sofware's *C4*[1] engine in our course, with projects similar to project 1 (Section 4.1) and project 3 (Section 4.3). The downside of using such a large code base and API is students often come away with less of a full understanding. Moreover, there is a steep learning curve to get even partial understanding, making it difficult for students that are slow to start to ever master the material in the course. While using open source game engines, such as *Grit*,[2] may have appeal due to their costs (free), the downsides are the same as for commercial engines since the issue is really about "production quality" engines and not whether or not they are freely distributed.

8 Conclusions

Computer science students' passion for games can be harnessed in many ways. Game-centric projects provide a ready motivator for students to explore and master difficult computer science topics. While there are many examples of educators having students build games or build parts of games to learn computer science topics, there are fewer that expose students to the programming richness of a game engine. Development of a game engine, a code base that supports making computer games, provides opportunities for strengthening programming skills with software patterns, object inheritance, real-time events handling, data structures, I/O debugging and testing and performance tuning. Game developing using an engine re-enforces this knowledge and provides portfolio material valuable for post-graduation employment.

This paper describes our Dragonfly approach, where students program a text-based game engine from scratch. Implementation projects proceed in phases, where students first build a stock game using our engine, proceed to develop their own engine through three sub-projects, then use their engine for a game of their own creation. Dragonfly has been used in two course offerings thus far, providing results for over 35 students.

Most students were able to successfully build a complete the Dragonfly] game engine, with about a quarter of them having excellent, full-featured game engines. These results suggest the objectives of the course, for students to understand game engines and apply software engineering techniques to their development, was successful. The majority of games made with the engine were good, suggesting the objective of students understanding a game engine from the game programmer's perspective was met. Student responses indicated they learned a lot from the course and greatly improved their C++ skills in making their game engines and games.

Continuing our efforts, the intent is to complete a book on Dragonfly (a draft is complete), suitable for classroom use, but also useful for students that may want to learn to build a game engine on their own. The structure of the book proceeds much as does the class material, providing details on the general functions of a game engine, with specific design details pertaining to Dragonfly. Once published, the book will include access to the Dragonfly tutorials, sample games and utilities and documentation on the engine as aids for students doing project 1 or project 3. Documentation generated by Doxygen is already online.[1] For instructors,[2] the complete Dragonfly source code is provided, with instructions on how to build the engine.

[1] `http://www.terathon.com/c4engine/`
[2] `http://gritengine.com/`
[1] `http://web.cs.wpi.edu/~claypool/papers/dragonfly-projects/include/`
[2] Some form of verification will be required.

References

Chaffin, A., Doran, K., Hicks, D., & Barnes, T. (2009). Experimental Evaluation of Teaching Recursion in a Video Game. In Proceedings of the ACM SIGGRAPH Symposium on Video Games, New Orleans, LA, USA (pp. 79–86). New York, NY, USA: ACM.

Claypool, K., & Claypool, M. (2005). Teaching Software Engineering Through Game Design. In Proceedings of the Innovation in Technology in Computer Science Education (ITiCSE) Conference, Jun. Lisbon, Portugal.

Coleman, R., Roebke, S., & Grayson, L. (2005). Gedi: a Game Engine for Teaching Videogame Design and Programming. *Journal of Computing Sciences in Colleges*, *21*(2), 72–82.

de Sousa, B.M.T. (2002). *Game Programming All in One*. Muska and Lipman/Premier-Trade.

El-Nasr, M.S., & Smith, B. (2006). Learning through Game Modding. *Computers in Entertainment (CIE)*, *4*(1).

Ganovelli, F., & Corsini, M. (2009). eNVyMyCar: A Multiplayer Car Racing Game for Teaching Computer Graphics. *Computer Graphics Forum*, *28*(8), 2025 – 2032.

Gestwicki, P., & Sun, F.S. (2008). Teaching Design Patterns Through Computer Game Development. *Journal of Educational Resources in Computing (JERIC)*, *8*(1), 2:1–2:22.

Gregory, J. (2009). *Game Engine Architecture*. AK Peters ISBN: 1-5688-1413-5.

Irvine, M. (2008). , Survey: 97 Percent Of Children Play Video Games. Huffington Post.

Kurkovsky, S. (2009). Can Mobile Game Development Foster Student Interest in Computer Science?. In Proceedings of the International IEEE Consumer Electronics Society's Games Innovations Conference, aug (pp. 92–100).

Muratet, M., Torguet, P., Jessel, J.P., & Viallet, F. (2009). Towards a Serious Game to Help Students Learn Computer Programming. *International Journal of Computer Games Technology*, (pp. 3:1–3:12).

Parberry, I., Nunn, J., Scheinberg, J., Carson, E., & Cole, J. (2007). SAGE: A Simple Academic Game Engine. In Proceedings of the Second Annual Microsoft Academic Days on Game Development in Computer Science Education, Feb. (pp. 90–94).

Thorn, A. (2010). *Game Engine Design and Implementation*. Jones and Bartlett Publishers ISBN: 0-7637-8451-6.

Vanhatupa, J.M. (2011). Game Engines in Game Programming Education: Experiences from Use ff the CAGE Game Engine. In Proceedings of the 11th Koli Calling International Conference on Computing Education Research, Koli, Finland (pp. 118–119). New York, NY, USA: ACM.

Volk, D. (2008). How to Embed a Game Engineering Course into a Computer Science Curriculum. In Proceedings of the Conference on Future Play: Research, Play, Share, Toronto, Ontario, Canada (pp. 192–195). New York, NY, USA: ACM.

Wang, A.I. (2011). Extensive Evaluation of Using a Game Project in a Software Architecture Course. *Transactions on Computing Education (TOCE)*, *11*(1), 5:1–5:28.

Yuzwa, E. (2006). *Game Programming in C++: Start to Finish*. Charles River Media ISBN: 1-5845-0432-3.