Verified Gaming

Joseph R. Kiniry
IT University of Copenhagen
Copenhagen, Denmark
kiniry@acm.org

Daniel M. Zimmerman
University of Washington Tacoma
Tacoma, Washington, USA
dmz@acm.org

ABSTRACT

In recent years, several Grand Challenges (GCs) of computing have been identified and expounded upon by various professional organizations in the U.S. and England. These GCs are typically very difficult problems that will take many hundreds, or perhaps thousands, of man-years to solve. Researchers involved in identifying these problems are not going to solve them. That task will fall to our students, and our students' students. Unfortunately for GC6, the Grand Challenge focusing on Dependable Systems Evolution, interest in formal methods—both by students and within computer science faculties—falls every year and any mention of mathematics in the classroom seems to frighten students away. So the question is: How do we attract new students in computing to the area of dependable software systems?

Over the past several years at three universities we have experimented with the use of *computer games* as a target domain for software engineering project courses that focus on *reliable systems engineering*. This position paper summarizes our experiences in incorporating rigorous software engineering into courses with computer game projects.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—Formal Methods, Programming by contract; K.3.2 [**Computers and Education**]: Computer and Information Science Education—Computer science education; K.8.0 [**Personal Computing**]: General—Games

General Terms

Design, Verification

Keywords

Grand challenges, formal methods, games in education

GAS '11, May 22, 2011, Waikiki, Honolulu, HI, USA

1. INTRODUCTION

According to conventional wisdom, game development and formal methods do not mix. Metaphors involving oil and water come to mind. In our experience, much of the game industry shuns classically-trained computer science students and instead looks for young self-trained hackers, and the standard examples in formal methods textbooks are either semi-trivial data structures or toy safety-critical systems.

Given the cost of developing modern console and online games, however, rigorous development techniques are quite valuable to the game development industry. Shipping a non-indy game today takes dozens of people and millions of dollars. This effort is comparable to that of developing and shipping a CPU in the early 1990s. A single serious bug encountered by many reviewers, or an unfortunate "Easter egg" leaked at the wrong time, effectively kills a game in the marketplace or damages a studio's reputation.

Conversely, experience has made it quite clear to us that the use of games in software engineering classes is an excellent motivator for students and an extremely effective vehicle for demonstrating and exercising formal techniques.

We believe that integration between formal methods and game development positively effects both areas of study. In this abstract, we provide an overview of the "emulsifying agent" that has let us successfully mix these two disparate domains in our software engineering classes.

2. GAMING IN PROJECT COURSES

After teaching project-centric courses for fifteen years, we have found that two things engage students more than anything else: hardware and games.

When we give a student a device to interact with—a smart card, a fingerprint reader, a Nintendo DS, an iPhone, or a LEGO MindStorms kit—the student has a physical manifestation of computing that turns the abstract nature of programming into an activity that controls reality.

Similarly, when we let students play games in the lab in the name of "performing domain analysis," they transform from gaming zombies into reflective learners. Students feel like they are getting away with something when, in fact, they are experiencing active engagement. This engagement is the first necessary step to opening their minds to formal methods, as deep analysis requires deep interest.

Moreover, when we give programming assignments that involve designing and building games, students seem to care significantly more about the quality of their work. This investment includes not only the basic functionality of a game, but also more artistic attributes such as soundtracks and

graphical assets. These facets permit students to express themselves in ways that many programming assignments do not. Many of our students have continued working on their projects even after the end of a course, showing them off to family and friends and occasionally even making them publicly available on the Web or in app stores.

2.1 Secret Ninja Formal Methods

A key to our success in combining formal methods and game development has been our use of *secret ninja formal methods* [1], a set of techniques that enables students to perform formal analysis, design, and development without being told that what they are doing is "formal" or "mathematical." They perform their analysis and design in a notation that appears to be merely structured English, which is then transformed (either by hand or automatically by our tools) into implementation skeletons with assertions (preconditions, postconditions, invariants, etc.). The resulting implementations are statically checked throughout the development process using multiple tools, and tested using automatically-generated unit tests, without the students doing anything more complex than saving their work in their IDE or clicking a "build" button.

Only after the students have successfully implemented one or more software projects do we reveal that, in fact, they have been using formal methods throughout the software development process. By then they have experienced what modern formal methods can accomplish and almost uniformly appreciate the value of the formal techniques.

2.2 Running Systems as Specifications

An important aspect of our approach is the use of existing games as "specifications" for student projects. The students pick, or are given, a game to replicate. They then play the game extensively in the lab, sometimes using an emulator such as MAME or VICE,[1] to discover various aspects of its design and functionality, generate their own analysis based on their observations, and (using the methods described above) implement their own version of the game in a high-level programming language.

Students perform experimentation games in the lab, exploring usage scenarios for real, rather than on paper or in their minds' eyes. When they encounter bugs or Easter eggs, such as a kill screen or the famous Konami Code,[2] discussions ensue about the extent to which these bugs and features should be replicated. Once they have thoroughly explored a game, they produce a concept analysis in our structured English notation. This analysis must be detailed enough to replicate the important aspects of gameplay, which includes all the game rules and constraints that ensure game balance as well as detail about graphics, sound, controls, etc.

As an example, consider the classic arcade game *Asteroids*. A reasonable concept analysis for *Asteroids* would contain control and gameplay characteristics: the ship is controlled by rotation and acceleration (rather than by moving linearly on the playfield); momentum is not conserved; there is an ominous background soundtrack that accelerates in tempo as asteroids are cleared from the playfield; there are three sizes of asteroid and all but the smallest breaks into two smaller pieces when shot; there are a small number of different asteroid shapes; asteroids can not collide with each other but can collide with flying saucers; and more.

Such an analysis would also contain several constraints related to game balance: at most four shots can be on screen at a time; the "hyperspace" action results in the ship's instant destruction on reentry approximately 25% of the time (as students would determine experimentally); and that large flying saucers fire randomly while small flying saucers target the ship with deadly accuracy. A reimplementation missing one of these constraints would not provide the same gameplay experience. Even a simple change, like allowing the player to have eight shots at a time on screen, would make the game significantly easier.

Having students replicate an existing game, rather than implement a game described on paper, has two main benefits. The most important of these is that the students are able to focus on the software engineering process rather than on the game design process. Formulating game mechanics and balancing them to provide a fulfilling play experience involves many issues that are tangential to the actual implementation and verification of the software, and our courses are not designed to specifically teach game development.

The other main benefit arises because most of the games the students use as specifications were originally implemented in dedicated arcade cabinets or on systems such as the Commodore 64. The original hardware of these games was far less capable than current target machines, and games were written in lower-level languages than those used in our courses. These differences help students directly connect with and reflect upon issues of resource utilization and performance, an understanding of which is invaluable in creating quality software. A reimplementation of a game that was written in assembly, distributed on a 4K ROM, and required 32K of RAM at runtime often takes thousands of lines of Java code and uses dozens of megabytes of RAM at runtime. Witnessing firsthand through emulation the incredible things that classic games do with such minimal resources forces students to carefully consider their own design and algorithm choices and to internalize the idea that high level languages incur costs as well as conferring benefits.

3. TEACHING EXPERIENCES

In this section, we summarize the range of courses we have taught using these methods, the evolution of our techniques, the manner in which we assess students, and the technologies we use.

3.1 Summary of Courses

The initial course that we co-taught with a gaming component was a third-year distributed systems course at Caltech in 1996. Students had to design and implement a distributed poker system in Java, including AI. We required the students to design a peer-to-peer architecture rather than a standard boring client-server architecture. Consequently, reasoning about the correctness of the distributed algorithms and the security of the system to prevent cheating was paramount. Students' poker bots competed in an extensive automated round-robin tournament at the end of the quarter to demonstrate the correctness and quality of their poker systems.

This class represents one end of a spectrum of students and goals. But, while Caltech students are second-to-none, at the time we had few tools to support a formal devel-

[1] Information about these emulators and the development tools that we discuss subsequently, and downloadable versions of most of them, can be found through http://www.verifiedgaming.org/.

[2] ▲▲▼▼◀▶◀▶ⒷⒶ, http://wikipedia.org/wiki/Konami_Code

opment process. Students hand-wrote and reasoned about their systems, sketched their architectures using drawing tools like `xfig`, and were only able to test their systems using manually written assertions and unit tests. Java did not yet have an `assert` statement, no standard logging frameworks existed, the JML tool suite did not yet have a working runtime assertion checker, and ESC/Java did not yet exist.

More recently we have taught students with a broad range of age and experience, from first-year students with only one semester of programming to seasoned software professionals from industry pursuing MSc degrees. This breadth in student background is evidence for our claim that these techniques—and indeed modern applied formal methods in general—do not require brilliant, experienced students.

The courses in which we use gaming as a "hook" also vary. While all are mandatory courses in our curricula, their titles include: *Software Engineering 1, 2, and 3*; *Analysis, Design, and Software Architecture with Project*; *Advanced Models and Programs*; *Distributed Systems Laboratory*; *Programming Practicum*. While injecting a game project into any systems course requires some creativity, we have even found ways of finding fun into courses as historically dry as *Foundations of Computing* (!).

Technologies. In some courses, students must *not* use Microsoft Windows, thereby gaining first-hand experience with alternative platforms (Linux, Mac OS X, iOS, Android) and their platform-specific (e.g., Core Video and Core Audio) and platform-generic, possibly open source, libraries (e.g., OpenGL, OSS, SDL, etc.) and development tools (e.g., Eclipse, XCode). In other courses, students *must* use Microsoft technologies (e.g., Visual Studio, XNA, etc.).[3] The choices of platforms, programming languages, and IDEs all have a direct impact on the content of our courses. That being said, every mainstream programming language today has assertions and a Design by Contract library, and most have some static analysis tool support.

For example, with Java we use the JML tool suite, OpenJML, BONc, Beetlz, ESC/Java2, FindBugs, PMD, CheckStyle, Metrics, JUnit, log4j, and the JDK logging and concurrency frameworks. With C# we use .NET's assertion and logging frameworks, Visual Studio's static checkers, and MSR's Code Contracts compiler, static checker, and PEX.

Special Considerations for Group Work. For group work, teams range in size from two to eight students. More mature students are permitted or encouraged to attempt to organize larger teams. Students take responsibility for specific subsystems of the project depending upon their interest and expertise. Students typically self-identify roles and self-organize into teams. We assist them to ensure that each team contains appropriate broad coverage of the domains necessary to accomplish the game development goals.

Summative and Formative Feedback. Typically, each team works with a dedicated teaching assistant who meets with and evaluates individual and group performance on a weekly schedule. Regular feedback is given during analysis, design, and development in two ways, online and offline. Online feedback is automated and interactive and uses our AutoGradeMe tool [2] and artifact reviews. Asynchronous offline feedback is given by TAs and instructors, watching RSS feeds of version control repositories and using collaborative development environments like Trac. Consistency of feedback is regulated and maintained by total or random double-assessment by TAs and the instructor.[4]

Concepts, Tools, and Technologies Covered. The topics covered include a subset of the following, depending upon the nature and length of the course and the maturity of the students:[5] UNIX; build systems; version control systems; documentation; *coding standards*; *metrics*; *assertions* and *contracts*; *specifications*; *system, integration, and unit testing*; collaboration/teamwork and collaborative development; use of collaborative development environments; *concept analysis*, *design*; patterns; *software architecture*; and *test generation*.

3.2 Student Projects

Over the years, our students have chosen many different games as projects. For the interested reader, we have archives of version control repositories going back half a dozen years that include hundreds of example projects.

Typical student projects are based on classic arcade titles (e.g., *Asteroids*, *Defender*, *Missile Command*, *Space Invaders*, *Tetris*), Commodore 64 games (e.g., *Elite*, *Space Taxi*, *Thrust*), or current "casual" games (e.g., *Flow*, *Plants vs. Zombies*) with which students are already familiar. Some students design vehicle simulators (inspired by titles like *Red Baron* and *Super Sprint*) or, if they are less interested in video games in general, real-world games such as *Monopoly* and poker. We have sometimes been surprised by the genres and technologies that are unpopular in our classes.

Genres. With regard to genre, action games are king. Adventure, role playing, simulation (construction and management, life, God games), real-time strategy, music, pure puzzle (as compared to action puzzle games like *Tetris*), and sports genres are nearly unrepresented. Likewise, and perhaps most surprisingly given the historical impact of the genre, platform games (e.g., *Jumpman*, *Pitfall!*, and of course *Super Mario Bros.*) are also unpopular.

We are unsure why the non-action genres are so unpopular. One possible explanation is that most students are more interested in the animation and rendering aspects of game implementation (and the resulting instant gratification of seeing their creations shoot bullets everywhere and explode in creative ways) than in the level design, puzzle creation, and story creation that are core components of non-action games. Such content creation is time-consuming and often viewed as unrewarding in programming-centric courses.

Technologies. Attempts at teaching the students to render with vector graphics (e.g., *Gravitar*, *Star Wars*, and *Tempest*) are often stymied by students' (and APIs') focus on bitmaps and sprites. Isometric games like *Congo Bongo* and *Zaxxon* are also unpopular, perhaps because tiling and character graphics are unfamiliar and under-appreciated.

When unconstrained, students often try in their naïveté to use complex modern technologies rather than simple and elegant classical techniques. For example, they are quite willing to dive into programming multiplayer network games with no background in distributed systems, using heavyweight constructs like Java RMI where datagrams or TCP sockets are all they need. Additionally, they typically attempt to use unstructured concurrency—instead of the tried-and-true event loop—to drive animation and process input.

[3] The former constraint is put in place by we instructors for the good of our students; the latter is placed upon us by our administrations.

[4] Scaling these multiple assessments has only recently become tractable with the introduction of AutoGradeMe.

[5] The topics that include subtle formal content are *emphasized*.

Motivations for Project Choice. As mentioned earlier, we usually ask students to mimic 8-bit games on modern hardware with modern programming languages and environments. This task sounds trivial to most students until they look closely at the behavior, pace, and depth of these classic titles. They quickly learn that gigahertz and gigabytes are far from necessary and sufficient to implement a great game.

Beyond challenging students' preconceptions about modern computing power, we must also convince them to stretch their proverbial legs. The project specifications we provide often include requirements that force students to learn and apply useful technologies. The following are examples of such requirements: players must register with the game and high scores must be persistent; a basic website for the game must be created and the game client must communicate with it to obtain updates; no multi-threading is permitted; all communication between subsystems must use protocols consisting solely of well-defined ASCII streams; the game's core event loop must be specified as a finite state machine.

System Decomposition. When performing co-analysis with students in the classroom, we often highlight key functional aspects of classic games that make for easier system decomposition and teamwork. Consequently, the members of a standard three person team typically tackle core data structures, I/O, and graphics/sound.

As previously mentioned, we often cajole or force students to define text-based communication protocols between subsystems. Defining and reasoning about such protocols opens up opportunities for the introduction of tools such as the UPPAAL model checker. Additionally, such interfaces lend themselves to subsystem testing via mock systems constructed using abstract state machine-based techniques. These basic interfaces also let us argue that the front-end of a video game—primarily its graphics and sound—is the easiest thing to create and should be done last.

What to Validate? What to Verify? Even small computer games are larger than most formal methods case studies in industry and academia. The smallest games our students typically submit are over 2,000 statements long. Consequently, teams must make critical decisions about where and how to focus their attention, given the limited time they can devote to the project.

What parts of the system should be validated? How should validation be accomplished? Via code reviews? Handwritten tests? Automatically generated tests? What parts should be verified? How should they be verified? Model checking? Abstract interpretation? Symbolic execution? Extended static checking?

There is no one right answer to these questions, of course. The important thing is that the students *ask* these questions, not necessarily *answer* them. Of course, through years of experimentation we have a set of best practices that we suggest to student teams, but these are descriptive rather than proscriptive features of our methodology.

4. REFLECTIONS

Does this technique work? Do the challenges inherent in getting a game *right* convince students that writing quality software necessitates tools and practices that are uncommon in industry? To our knowledge, no one has ever before attempted to write a verified computer game, let alone to convince several generations of students to attempt such. We believe that our approach is promising, and that both formal methods and game development can benefit from this sort of integration. However, all is not roses. There are several challenges in teaching this material, and in engaging with students, that we have not yet overcome.

4.1 Open Challenges

First, students prefer polish over correctness. They would much rather spend the final hours of a project on animating zombies than on fixing class invariants. This preference is unsurprising, as avoiding hard work for easy work is something virtually everyone does. We have some evidence that this issue can be corrected through refinements in assessment methods. For example, as is partially possible with AutoGradeMe, biasing good grades towards high specification and test coverage and low numbers of moderate-to-serious static analysis warnings is a reasonable approach.

Second, formally specifying some of the standard architectural styles inherent in game programming, particularly sense-compute-control and implicit invocation (aka events and callbacks), is an open research challenge. It is always interesting to see what tricks students come up with to attempt to reason about these parts of their implementations.

Third, many teams choose (not unwisely) to implement their games using existing rich APIs like XNA or Java2D. Unfortunately, these APIs have no formal specifications, so formally reasoning about code that uses them is impossible.

Finally, students find testing games, beyond writing or generating unit tests for core data structures, quite conceptually difficult. Our speculation is that this difficulty comes from the introduction of input, rendering, and timing requirements. Testing a core data structure is essentially independent of these; if you provide data X to operation O you expect to get result Y, and while termination of the operation is important, elapsed time is typically not. By contrast, a test of a bullet flying across a screen may involve both rendering requirements (is it being drawn properly?) and timing requirements (is it moving with the correct velocity?). Likewise, a test of an on-screen character responding to player input may involve rendering requirements, timing requirements, and input requirements.

Perhaps if more class time were dedicated to exactly these challenges and their potential solutions, students would take validation more seriously than their standard answer, "ship it to thousands of testers and let them give feedback." Our first experiments with validating and verifying game event loops and rendering via a "Verified Pong" implementation written by two MSc students are promising.

4.2 Pedagogical Resources

As mentioned earlier, we have archives of version control repositories that include many example projects. We are posting these projects, course slides, demo games, and supporting materials, as well as information about the tools we have discussed, to `http://www.verifiedgaming.org/`.

5. REFERENCES

[1] Joseph R. Kiniry and Daniel M. Zimmerman. Secret ninja formal methods. In *Fifteenth International Symposium on Formal Methods (FM)*, 2008.

[2] Daniel M. Zimmerman, Joseph R. Kiniry, and Fintan Fairmichael. Toward instant gradeification. In *24th IEEE-CS Conference on Software Engineering Education and Training (CSEE&T)*, to appear, 2011.