Verified Gaming

Joseph R. Kiniry
IT University of Copenhagen
Copenhagen, Denmark

kiniry@acm.org

Daniel M. Zimmerman
Univ. of Washington Tacoma
Tacoma, Washington, USA

dmz@acm.org

1st Games and Software Engineering Workshop - 22 May 2011

Our Assumptions

- formal methods play a critical role in the development of reliable software systems
- students in programming courses tend to run away screaming when confronted directly with the associated mathematics
- game-related projects are excellent motivators for students

The Core Ideas

- if students successfully implement a game using a formal methods-rich process, they will have fun and also learn to appreciate formal methods
- since games exhibit complex behavior relative to implementation size, this is an excellent way to exercise and improve our formal development tools and techniques

A Related Idea

- applying lightweight formal methods to real-world game development can have substantial benefits, since the cost of failure on game projects can be quite high
- we are pursuing work along these lines, but so far we have focused more on the benefits games can provide for formal methods development than vice-versa

Achieving Verified Gaming in Courses

- *secret ninja formal methods* – a formal development process that doesn't frighten students away
- *running systems as specifications* – existing games as the basis for class projects

Secret Ninja Formal Methods

- we incorporate formal methods into the development process with *minimal new notation* – i.e., *stealth mathematics*
- our students know *English* and *Java*, so we use them for *informal* and *formal* specifications (all in a formal framework)
- we *align learning with engineering* by coupling assessment with tool feedback

Secret Ninja Process

- multiple stages, all *reversible*
  - analysis and design in structured English (informal BON)
  - refinement to implementation skeletons with assertions (JML, looks like Java)
  - "filling in the blanks"
- continuous static checking and automated testing throughout development

Concept Analysis

- agree upon the (domain) concepts
  - Weapon, Shuriken, Point, Velocity, Enemy
- define each with a simple English statement
  - Shuriken - “a weapon in the form of a star”
- identify all *is-a* and *has-a* relations
  - Shuriken *is-a* Weapon
  - Shuriken *has-a* Point

Describe Concepts

- identify queries, commands, and constraints
  - Shuriken...
  - How many points do you have?
  - Fly toward that enemy!
  - You must have at least three points.

Capture Specs in BON

```
class_chart SHURIKEN
  inherit WEAPON
  indexing
    author: “Secret Ninjas”
  description
    “a weapon in the form of a star”
  query
    “How many points do you have?”
  command
    “Fly toward that enemy!”
  constraint
    “You must have at least three points.”
end
```

Refine Informal BON into Documented Types

```
/**
 * A weapon in the form of a star.
 *
 * @author Secret Ninjas
 */
class Shuriken extends Weapon {
  /** How many points do you have? */
  /** Fly toward that enemy! */
  /** You must have at least three points. */
}
```

Introduce Signatures

```
/**
 * A weapon in the form of a star.
 *
 * @author Secret Ninjas
 */
class Shuriken extends Weapon {
  /** How many points do you have? */
  byte points();

  /** Fly toward that enemy! */
  void attack(Enemy the_enemy);

  /** You must have at least three points. */
}
```

Specs

```
/**
 * A weapon in the form of a star.
 *
 * @author Secret Ninjas
 */
class Shuriken extends Weapon {
  /** How many points do you have? */
  /*@ pure */ byte points();

  /** Fly toward that enemy! */
  //@ ensures the_enemy.slain();
  void attack(/*@ non_null */ Enemy the_enemy);

  /** You must have at least three points. */
  //@ invariant 3 <= points();
}
```

Running Systems as Specifications

- students pick, or are given, a classic game to replicate
- they then (exhaustively!) play the *original* game in an emulator like MAME or VICE
- the goal: discover how the game works (rules, constraints, balance, bugs, ...), generate an O–O analysis and design, and implement it in a high-level language

Running Systems as Specifications

- why have students implement games that already exist, rather than design their own?
  - they can focus on software engineering concepts rather than on game design
  - they can see issues of resource utilization, performance, etc. first-hand – classic games are extremely impressive despite minimal computing resources

Example

- one class project used *Thrust*, a C=64 game where the player pilots a ship in a cave to pick up pods and fly them into space

Example

- there are a number of different entities in the game
- the ship is subject to gravity and inertia, so physics comes into play
- BON charts

Results

- students are generally excited about the courses in which we use this technique
- the resulting games are mostly reasonable reimplementations, though they aren't always completely finished within an academic quarter/semester
- we've had some success with validation and verification of game event loops and rendering ("Verified Pong" MSc project)

http://www.verifiedgaming.org/

- links to everything we use (tools, techniques, course pages, etc.)
- more will become available over time, including student projects – some are already available from linked course pages
- questions?

Domo Arigato