Writing Real-Time .Net Games in Casanova

Giuseppe Maggiore, Pieter Spronck, Renzo Orsini, Michele Bugliesi,
Enrico Steffinlongo, and Mohamed Abbadi

Università Ca' Foscari Venezia
DAIS - Computer Science
{maggiore,orsini,bugliesi,esteffin,mabbadi}@dais.unive.it
Tilburg University
Tilburg Center for Creative Computing
p.spronck@gmail.com

Abstract. In this paper we show the Casanova language (and its accompanying design pattern, Rule-Script-Draw) in action by building a series of games with it. In particular we discuss how Casanova is suitable for making games regardless of their genre: the Game of Life, a shooter game, an adventure game and a strategy game. We also discuss the difference between Casanova and existing frameworks.

Keywords: Game development, Casanova, databases, languages, functional programming, F#.

1 Introduction

There is a growing, substantial interest in research on principled design techniques and on cost-effective development technologies for game architectures. This is driven by the diffusion of independent games, an increased need for fast prototyping gameplay mechanics [1], and the need to develop serious or research games [2] for which the same budget of blockbuster titles cannot be spared. Moreover, as the games market keeps growing in size [3] this need is further emphasized. Our present endeavor makes a step along the directions of studying disciplined models for game development.

In this paper we discuss the Casanova language for making games. We do not present Casanova as such, that being the focus of other papers [5,6,7]. Rather, given that Casanova exists and works already (albeit under a prototypical implementation), we study and measure its feasibility when used for making games. We will thus try and answer the research question *"does Casanova make game development easier?"* by identifying a series of general, orthogonal activities in game development that we show built in Casanova, and we will compare our implementation with other languages. We start with a discussion of related work in section 2. We give a first description of Casanova in section 3. We discuss with detailed examples how to make actual games with Casanova in section 4. In section 5 we compare Casanova, C# and F# when used for the games of the preceding section. In sections 6 and 7 we conclude by

M. Herrlich, R. Malaka, and M. Masuch (Eds.): ICEC 2012, LNCS 7522, pp. 341–348, 2012.

discussing some of the extensions that we are planning on adding to Casanova with our future research.

2 Related Work

To build the logic of a game the two most common software architectures are object-oriented hierarchies and component-based systems. In an object-oriented engine the hierarchy is rooted in the `Entity` class [9]. A component-based system defines each game entity as a composition of components that provide reusable, specific functionality [10]. These two more traditional approaches both suffer from a noticeable shortcoming: they focus exclusively on representing single entities and their update operations in isolation: each entity needs to update itself at each tick of the simulation. This shifts the game focus away from the interactions between entities (collision detection, AI, etc.), from which most of a game complexity comes. Another particularly nasty problem that arises in traditional game development is that of representing long-running behaviors of entities; such behaviors are those processes performed by game entities which last many ticks of the game loop to complete. These behaviors are coded as explicit state machines inside the entities, thereby forcing entities to store spurious data that does not have to do with the entity logical model but rather with the representation of its state machines.

Alternative paradigms have been experimented as part of various research efforts: functional reactive programming (FRP, see [11]), a data-flow approach where values are automatically propagated along a dependency graph that represents the computation; and automatically optimized SQL-queries for games (the SGL language, see [12]). FRP mitigates the problem of representing long-running behaviors, but it offers little else to game development, while SGL focuses exclusively on defining the tick function and not on representing long-running processes.

We have designed Casanova around all these issues: Casanova promotes entities that interact with each other, queries on the game world, long-running behaviors, automated drawing of the game entities, and even consistency of the game world.

3 The Casanova Language

The Casanova language belongs to the ML family (F# in particular, with list comprehensions inspired from the elegant Haskell syntax). Its main design focus is syntactic simplicity, where the language is built around few linguistic primitives that are powerful enough to be combined into many games.

A Casanova game begins with the definition of a series of data structures, which are the world and its entities. The updates of an entity are contained in its rules, a series of methods that take the same name of the field they update at each tick; a rule is invoked automatically for each entity of the game, and it receives as input the current state of world, the current state of the entity being updated, and the time delta between the current frame and the previous frame. All rules do not interfere with each other, can be computed in parallel, and exhibit *transactional* behavior; this avoids temporal inconsistencies where the game world is partially updated. Entities may also have drawable fields such as text, sprites or 3D models; these fields are updated

through rules, and at each tick all drawable entities are grouped into *layers* (layers specify a series of draw settings) for drawing.

We could define a hypothetical game world as a series of balls. The world also features a sprite layer, to which renderables will be assigned (Listing 1).

```
type World =   { Sprites     : SpriteLayer
                 Balls       : var<list<Ball>> }
```

Listing 1. A world of balls

Each ball (Listing 2) contains a position and a velocity, in addition to a sprite for drawing. The position is updated by moving it along the velocity, and the sprite position is taken from the entity position:

```
type Ball = {
    Position    : Vector2<m>
    Velocity    : Vector2<m/s>
    Sprite      : DrawableSprite {
  rule Position(world,self,dt) = self.Position + dt * self.Velocity
  rule Sprite.Position(world,self,dt) = self.Position
```

Listing 2. Ball

The initial state of the game features no initial balls and the empty sprite layer. We omit this listing as it is rather straightforward.

We use the main script of the game to create random balls every few seconds (Listing 3).

```
let main world =
  repeat { wait 1.0
           world.Balls.Add … }
```

Listing 3. Spawning balls

4 Making Games with Casanova

To assess the effectiveness of Casanova as a game development language we have undertaken two parallel development initiatives. One such initiative is [13], where we have built a series of small samples that are easy to understand and manipulate; these samples are a series of real-time games, chosen so as to see Casanova in action in different sub-domains of the real-time game genre (possibly the most widespread nowadays). These samples are an asteroid shooter game, an action/adventure game and a strategy game. We will not present the full samples themselves, which are available online. We will now focus on a series of fundamental "development activities" that we believe to be nicely exemplified by our samples; these activities cover some of the most common and important pieces that can be customized, combined and extended into almost any game: *(i)* defining a *player avatar*, handling his input and his shooting; *(ii) spawning obstacles* randomly; *(iii)* handling *collisions* between projectiles and obstacles; *(iv) active entities* such as bases or buildings that produce units; *(v) selection-based input* mechanisms. We show *(i)*, *(ii)*, and *(iii)* from the asteroid shooter in 4.1, and *(iv)* and *(v)* are shown from the RTS game in 4.2. The primitives shown here are just samples, but they can be recombined, modified and reassembled into many new games; we also point the existence of [14], an upcoming (commercial) strategy

game that is derived from the RTS sample and that we are building as an ongoing study of how to create non-trivial games with Casanova with this extension process.

4.1 Player Avatar and Shooting Stuff

The asteroid shooter game is a simple shooter game where asteroids fall from the top of the screen towards the bottom and must be shot down by the player.

In this game we will describe how to define: *(i)* the *player avatar*, his movement and shooting; *(ii)* the *spawning of obstacles* such as asteroids; and *(iii)* detection of *collisions* between asteroids and projectiles.

The game world (Listing 4) contains a list of projectiles, asteroids, the cannon, the current score, plus sprite layers for the main scene and the game UI. The game world removes asteroids and projectiles when they exit the screen or collide with each other, and it handles the current score (which is the number of destroyed asteroids).

```
type World = {
  Sprites         : SpriteLayer
  UI              : SpriteLayer
  Asteroids       : var<list<Asteroid>>
  Projectiles     : var<list<Projectile>>
  Cannon          : Cannon }
  rule Asteroids(world,dt) =
    [a | a <- state.Asteroids && a.Colliders.Length = 0 && a.Position.Y < 100.0<m>]
  rule Projectiles(world,dt) =
    [p | p <- state.Projectiles && p.Colliders.Length = 0 && p.Position.Y > 0.0<m>]
```

Listing 4. Asteroids world

The player is represented by a cannon (similarly it might be represented by a moving ship) as an entity that contains a sprite, an angle, and two boolean movement flags set from the input script that determine the variation of the angle and which are reset to `false` at every tick; the rotation of the sprite is taken from the current angle of the cannon (Listing 5).

```
type Cannon = {
  Sprite          : DrawableSprite
  Angle           : float<rad>
  MoveLeft        : var<bool>
  MoveRight       : var<bool> }
  rule Angle(world,self,dt) =
    self.Angle + if self.MoveLeft then dt elif self.MoveRight then -dt else 0.0<rad>
  rule MoveLeft(world,self,dt) = false
  rule MoveRight(world,self,dt) = false
  rule Sprite.Rotation(world,self,dt) = self.Angle
```

Listing 5. Cannon

The input script that modifies the rotation of the cannon simply checks if the appropriate key is currently pressed, and if so the cannon movement values are set (Listing 6).

```
  { return is_key_down Keys.Left } => { state.Cannon.MoveLeft := true },
  { return is_key_down Keys.Right } => { state.Cannon.MoveRight := true }
```

Listing 6. Cannon movement

Similarly, projectiles are generated (or "spawned", Listing 7) whenever the space key is pressed; contrary to movement, though, after a projectile is spawned the script waits one-tenth of a second to ensure that projectiles are not shot with a frequency of

one per frame. It is worth noticing that such a simple activity would require a timer-based event infrastructure that can be quite tedious to write in a traditional language; for example, a timer to wait after the spawning of a projectile would need to be stored, declared, and consulted manually at each tick.

```
{ return is_key_down Keys.Space } => {
    state.Projectiles.Add …
    wait 0.1<s> }
```

Listing 7. Shooting

Asteroids are generated with a simple recursive script that waits a random amount of time and then adds the asteroid to the game world (Listing 8).

```
repeat {
    wait (random(1.0<s>,3.0<s>))
    state.Asteroids.Add …
}
```

Listing 8. Spawning asteroids

Collision detection is simple as well: both asteroids and projectiles compute the list of colliders against themselves; this list is then used in the query shown above in the definition of the game world to cull away asteroids (or projectiles) that are hit by other entities (Listing 9).

```
type Asteroid =
  …
  rule Colliders(world,self,dt) =
    [x | x <- get_colliders world && distance(self.Position, x.Position) < 10.0f]
```

Listing 9. Asteroids collision detection

4.2 Active Map Entities and Selection-Based Input

The strategy game features a series of planets that produce ships, which can then be sent to conquer other planets. In this game we can see: *(iv) active entities*, such as planets, that represent complex components of the game scenario; and *(v)* complex *selection-based input* mechanisms based on the selection of game entities and the interaction with the selected entities. We represent the game world as a series of planets, ships, plus the currently selected planet; the game world also contains sprite layers for rendering the game entities and UI, plus a boolean that allows the game battles to tick at fixed time intervals rather than at each tick of the game (Listing 10).

```
type World = {
  Sprites         : SpriteLayer
  UI              : SpriteLayer
  Planets         : list<Planet>
  Fleets          : var<list<Fleet>>
  TickBattles     : var<bool>
  SourcePlanet    : var<Option<ref<Planet>>> }
rule Fleets(world,dt) =
  [f | f <- self.Fleets && f.Alive && (not(f.Arrived) || f.Fighting)]
```

Listing 10. RTS world

Planets manage the battles in their orbit (which determine the owner of the planet) and ship production; in addition to their other fields, planets store the current owner, the number of allied ships stationed on the planet, and the percentage of production

for the next ship; furthermore, a planet maintains a list of the fleets that are targeting itself for attacking or defending it (Listing 11).

```
type Planet = {
  Owner             : Player
  … }
rule Owner(world,self,dt) =
    if self.Armies <= 0 && self.AttackingFleets.Length > 0 then
      self.AttackingFleets[0].Owner
    else self.Owner
…
```

Listing 11. Planet definition

Input scripts manage the selection of a new planet by waiting for a left click of the mouse and then setting the `SourcePlanet` field of the game world (Listing 12).

```
  { return mouse_clicked_left() } => {
      let mouse = mouse_position()
      let clicked =
         [p | p <- world.Planets && distance(p.Position,mouse) < 10.0 && p.Owner =
Human]
      if clicked <> [] then return Some(clicked.Head)
      else return None } => fun p -> { world.SourcePlanet := Some(p) },
```

Listing 12. Planet selection

Similarly, when the user right clicks if there is an active selection some ships are sent (Listing 13).

```
  { return mouse_clicked_right() && world.SourcePlanet <> None } => {
      let mouse = mouse_position()
      let clicked = [p | p <- world.Planets && distance(p.Position,mouse) < 10.0]
      if clicked <> [] then return Some(clicked.[0],world.SourcePlanet.Value)
      else return None } => fun (source,target) -> { mk_fleet source target }
```

Listing 13. Issuing orders

5 Final Assessment

Assessing the quality of a programming language for a given activity is a daunting task. It is with this in mind that we proceed with offering a series of arguments in answer to our original claim that Casanova is better suited than traditional languages such as C# for real-time game development. Casanova programs are overall much shorter than equivalent C# programs (measured excluding trivial constructs such as constructors or properties), as are all the analyzed snippets. We also include data from the same games implemented in straightforward F# together with our Casanova libraries, which allows retaining most of the advantages of Casanova. The first comparisons that we make can be seen in Figure 1, and is concerned with the surrounding infrastructure, which is all the game code that is not strictly part of the game logic or drawing, and the overall length of the various samples.

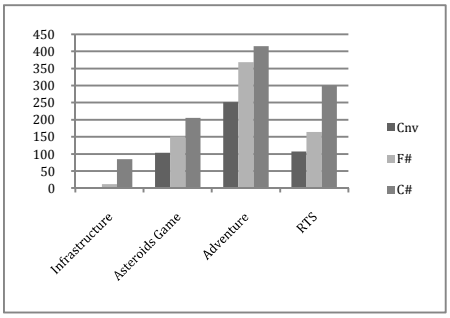

Fig. 1.

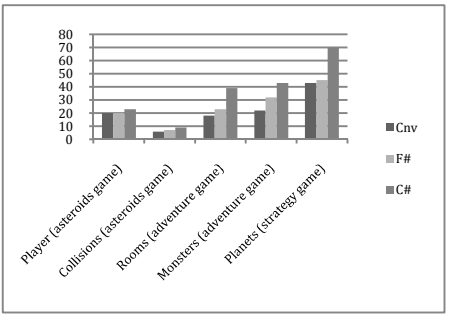

Fig. 2.

In Figure 2 we can see the comparison of the single snippets of game code that we have analyzed. With the data above, we feel it's safe enough to conclude that Casanova allows to express game-related concepts with less verbosity than traditional mainstream languages; specifically, Casanova completely removes the need for boilerplate code to initialize the game (since it is already built-in), and it removes the need to traverse the game world to update and draw each entity (since the framework takes care of evaluating rules and drawing drawable entities). The F# samples fare comparably to Casanova, both in terms of length and code complexity, mostly thanks to the Casanova library that allows the definition of scripts and automated traversal of the game state.

Finally, while we don't have enough data to qualify as a proper user study, we wish to point out the importance of our work in an actual game project, the upcoming strategy game Galaxy Wars [14]. This project is the complete version of the RTS sample discussed in the previous sections. Compared with the RTS sample, Galaxy Wars is much larger (tens of thousands of lines of code), and we have developed it both as a commercial endeavor and as a research test-bed for Casanova, with the aid of a group of Master students in Computer Science.

6 Future Work

We believe our work to have opened exciting new venues of exploration. Casanova started with the goal of making it simpler to build a declarative, easily optimized game logic, with its associated rendering. In addition to completing support for the Casanova language in terms of compiler, development tools, and visual editors, we will: *(i)* design further Casanova components such as menus, networking and audio systems; *(ii)* study a list of query optimizations [16] that could make Casanova more efficient; *(iii)* work on user studies on students and even actual game designers.

7 Conclusions

Game development is a large aspect of modern culture. Games are used for entertainment, education, training and more, and their impact on society is significant. This

is driving a need for structured principles and practices for developing games and simulations. Also, reducing the cost and difficulties of making games could greatly benefit development studios with less resources, such as independent-, serious-, and research-game makers. Casanova is a study in the automation and support of the most common game-development activities, in order to allow game developers to put more effort on what really matters (AI, gameplay, shaders, procedural generation, etc.) instead of smaller technicalities.

While Casanova is still in its early stages, we have used it extensively and with good results in a real game [14], and we are certain that with further work the benefits of this approach will become much more apparent.

References

1. Fullerton, T., Swain, C., Hoffman, S.: Game design workshop: a playcentric approach to creating innovative games. Morgan Kaufman (2008)
2. Ritterfeld, U., Cody, M., Vorderer, P.: Serious Games: Mechanisms and Effects (2009)
3. Entertainment Software Association: Industry Facts (2010)
4. Buckland, M.: Programming Game AI by Example, Sudbury, MA (2004)
5. Giuseppe Maggiore, M.: Monadic Scripting in F# for Computer Games, Oslo, Norway (2011)
6. Maggiore, G., Spanò, A., Orsini, R., Costantini, G., Bugliesi, M., Abbadi, M.: Designing Casanova: A Language for Games. In: van den Herik, H.J., Plaat, A. (eds.) ACG 2011. LNCS, vol. 7168, pp. 320–332. Springer, Heidelberg (2012)
7. Maggiore, G., Bugliesi, M., Orsini, R.: Casanova Papers. In: Casanova project page, `http://casanova.codeplex.com/wikipage?title=Papers` (accessed 2011)
8. DeLoura, M.: The Engine Survey. In: Gamasutra, `http://www.gamasutra.com/blogs/MarkDeLoura/20090316/903/The_Engine_Survey_Technology_Results.php` (accessed 2009)
9. Ampatzoglou, A., Chatzigeorgiou, A.: Evaluation of object-oriented design patterns in game development. Journal of Information and Software Technology 49 (2007)
10. Folmer, E.: Component Based Game Development – A Solution to Escalating Costs and Expanding Deadlines? In: Schmidt, H.W., Crnković, I., Heineman, G.T., Stafford, J.A. (eds.) CBSE 2007. LNCS, vol. 4608, pp. 66–73. Springer, Heidelberg (2007)
11. Conal, E., Hudak, P.: Functional reactive animation. In: International Conference on Functional Programming (ICFP), pp. 263–273 (1997)
12. Walker White, A.: Scaling games to epic proportions. In: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD), New York, NY, USA, pp. 31–42 (2007)
13. Maggiore, G.: Casanova project page (2011), `http://casanova.codeplex.com/`
14. Maggiore, G.: Galaxy Wars Project Page (2010), `http://vsteam2010.codeplex.com`, `http://galaxywars.vsteam.org`
15. Zhao, R., Szafron, D.: Generating Believable Virtual Characters Using Behavior Capture and Hidden Markov Models. In: van den Herik, H.J., Plaat, A. (eds.) ACG 2011. LNCS, vol. 7168, pp. 342–353. Springer, Heidelberg (2012)
16. Garcia-molina, H., Ullman, J., Widom, J.: Database System Implementation (1999)