Writing Real-Time .Net Games in Casanova

Giuseppe Maggiore, Pieter Spronck, Renzo Orsini, Michele Bugliesi, Enrico Steffinlongo, 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 discuss the difference between Casanova and existing frameworks, with particular focus on C#, F# and XNA, and we show a detailed comparison between them.

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.

Making games requires a large effort, in order to ensure that the final result stands up to the user expectations [4]; for this reason a game needs a high visual quality to clearly show the various states of the logical entities of the game. The logical entities of the game, in the meantime, are updated according to an articulated simulation that evolves their state in a meaningful way. The visual and logical modules of the game are large and complex to build; to make matters worse, both must run in a loop that is optimized enough to update the screen and animate the game entities in real-time, that is all iterations of the game logic and drawing must be completed in a time span that ranges between $1/20^{th}$ and $1/60^{th}$ of a second. As an additional challenge, game-making comprises a (rather large)

creative portion that is performed by designers, who rarely are well-versed in the arcana of computer programming: for this reason the architecture of a game must be flexible and easily modifiable so that designers can quickly build and test new iterations of gameplay. Games offer a unique blend of complexity, optimization, and need for customization by non-programmers; this makes games costly to build and maintain.

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?* Unfortunately, assessing the feasibility of a language at any given task is, to put it bluntly, impossible. What we do to approximate such an analysis is then slightly different: we will identify a series of general, orthogonal activities in game development (creating a player avatar, creating an active scenario, creating a monster with an AI, etc.) and we will show how to build them in Casanova; these activities come from different sample games that we have implemented in Casanova. We will then compare our implementation with other game development languages, to assess their verbosity and the amount of programming notions that come into play. 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 discussing some of the extensions that we are planning on adding to Casanova with our future research.

2 Related work

Building a rendering system in a modern game involves, at its core, the building of a scene graph that is fast to traverse for the retrieval of the visual entities of the game. Rendering engines are either built from scratch or licensed from other game studios. Building a new engine is a hard and time-consuming task that may involve a group of developers several years of constant effort. The apparently obvious solution of using an external rendering engine carries some important risks as well: engines are large and complex pieces of software that are hard to use effectively. Sometimes it may even prove difficult to adapt an existing engine to the evolving needs of the game. In fact, rendering engines tend to be relatively monolithic, that is when an engine is built for a specific type of game then it will be difficult to use for other genres [8].

When it comes to building the logic of a game, the two most common software architectures are object-oriented hierarchies and component-based systems. In a traditional object-oriented game engine the hierarchy represents the various game objects all derived from the general `Entity` class. Each entity is responsible for updating itself at each tick of the simulation [9]. A component-based system de-

fines each game entity as a composition of components that provide reusable, specific functionality such as animation, movement, reaction to physics, etc. Component-based systems are being widely adopted, and they are described in [10]. These two more traditional approaches both suffer from a noticeable shortcoming: they focus exclusively on representing single entities and their update operations. By doing so they lose the focus on the fact that most entities in a game need to interact with one another (collision detection, AI, etc.), and usually lots of a game complexity comes from defining (and optimizing) these interactions. One particularly nasty problem that arises in traditional game development is that of representing long-running behaviors of entities. Long-running behaviors are all those processes performed by game entities and which take many ticks of the game loop to complete. Usually, these behaviors are represented by handcrafting 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 implementation necessities of behaviors. Scripting systems are often attached to a game engine to mitigate the problem, but binding different languages introduces its own set of problems. Finally, these architectures simply upgrade everything in place, and offer no guarantees of consistency or against duplicated updates of an entity.

There are two additional approaches that have emerged in the last few years as possible alternatives to object-orientation: (functional) reactive programming and SQL-style declarative programming. Functional reactive programming (FRP, see [11]) is a data-flow approach where value modification is automatically propagated along a dependency graph that represents the computation. While FRP offers a solution to the problem of representing long-running behaviors, it neither addresses the problem of many entities that interact with each other, nor does it address the problem of maintaining the consistency of the game world. SQL-queries for games have been used with a certain success in the SGL language (see [12]). This approach uses a lightweight, statically compiled and heavily optimized query engine for defining a game. SGL suffers when it comes to representing long-running behaviors, since it focuses exclusively on defining the tick function.

We have designed Casanova with these issues in mind: with Casanova, the integration of the interactions between entities and long-running behaviors is seamless, the resulting game world is always consistent, and integrating the visual aspects of the simulation with visibility culling, a scene-graph, and shaders and effects can be done declaratively and at a smaller cost for the developer.

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. We now describe Casanova by showing how to build an application in it (3.1). The Game of Life,

while not properly a videogame (there is no interaction), features many of the aspects of a game: it is a simulation of a virtual world that evolves and changes over time. To help achieve some of the benefits of Casanova, but without the hurdles associated with using a young language, we also (briefly) discuss the design pattern that condenses the overall philosophy of Casanova (3.2). This design pattern can then be used in different languages, with varying benefits.

3.1 Game of Life

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. The result of computing a rule is stored in its entity only after all rules of all entities of the world have been computed successfully, that is rules do not interfere with each other and can be computed in parallel; this avoids inconsistencies deriving from the state being only partially updated: the state is either at a time step or at the next, but no in-between representations are allowed. 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) which are then drawn.

We start by defining the state of the game as a matrix of cells; the state also contains a boolean variable which will trigger the update of the cell matrix once per second. The world also features a sprite layer, a grouping of all the renderable sprites that specifies their common rendering parameters (Listing 1).

```
type World =
  { Sprites     : SpriteLayer
    Cells       : list<list<Cell>>
    UpdateNow   : var<bool> }
```

Listing 1 – Game of Life world

Each cell (Listing 2) contains a value (which is `1` when the cell is alive and `0` when the cell is dead) and a list of its neighbors (marked as `ref`, since the neighbors of a cell are just references to those cells, rather than the main points of storage for those cells, and thus they are not updated). The value of the cell is updated every time an update is triggered (rather than at each tick), by summing the value of the neighbors and applying the rules mentioned above. The color of the cell sprite is updated to reflect its current value.

```
type Cell = {
    NearCells   : list<ref<Cell>>
    Value       : int
    Sprite      : DrawableSprite {
  rule Value(world,self,dt) =
    if state.UpdateNow then
      let around = sum [c.Value | c <- self.NearCells]
      match around with
      | 3 -> 1
      | 2 -> self.Value
```

```
      | _ -> 0
    else self.Value
  rule Sprite.Color(world,self,dt) = if self.Value = 0 then Color.Black else Color.White
```

Listing 2 - Cell

The initial state of the game creates the matrix of cells and initializes their neighbors. Each internal cell has exactly eight neighbors. The sprites layer and the cell sprite are also setup for each sprite. We omit this listing as it is rather straightforward.

The rules of the game are fired at every frame of the game that is roughly 60 times per second. Changing the entire matrix of cells this often would yield a chaotic result; for this reason we have defined the `UpdateNow` field in the game state, so that we can control when the rules are fired. The main script of the Game of Life simply waits for a second before toggling the `UpdateNow` value, and then it suspends itself until the next iteration of the update loop. When the script is resumed, it toggles `UpdateNow` again and finally it repeats (Listing 3).

```
let main world =
  repeat {
      wait 1.0
      world.UpdateNow := true
      yield
      world.UpdateNow := false }
```

Listing 3 – `UpdateNow` timer

This way the game of life will run at exactly one step per second, no matter the framerate of the simulation. The game of life features no input, and thus no input script is specified.

3.2 The Rule-Script-Draw Pattern

We have experimented with the use of Casanova outside of the Casanova language through the RSD pattern for making games. RSD focuses on the description of a game as a combination of: *(i)* rules that specify how to update an entity *locally*; *(ii)* scripts that are imperative processes that modify the entire game state and which may suspend by `yield`-ing; and *(iii)* drawable entities which are updated through rules (so that the drawable fields reflect the current state of their associated logical entity) and which are then drawn.

Entities may be naturally implemented as classes, which rules are simply methods (static or instance methods) that act over fields that are double-buffered so as to avoid interferences. Scripts may be implemented with coroutines or threads (coroutines may be a better choice because threading systems easily get out of hand because of concurrency issues), either with monadic systems or with other meta-programming facilities. Drawing requires a scene-graph with a registration mechanism, so that when a drawable field is found on an entity during a tick then that field is registered to the scene graph which then draws it (notice that this mechanism allows the use of a separate rendering engine as long as it is made to provide these primitives).

Sources for existing RSD implementations may be found at [13]. The main implementation of the pattern is an F# library which features scripts through mon-

ads and rules invoked through (highly optimized) reflection. To work with a language which is farther away from the ML family (and widespread among game developers), we have also built an experimental C++ version of the same library, which makes heavy use of template meta-programming.

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 three different 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)* representing the properties of a static map (cave) divided in *rooms and cells*; *(v)* handling *monsters and their AI* (albeit a rudimentary one); *(vi) active entities* such as bases or buildings that produce units; *(vii) selection-based input* mechanisms. We show *(i)*, *(ii)*, and *(iii)* from the asteroid shooter in 4.1; *(iv)* and *(v)* are taken from the action/adventure game and shown in 4.2; finally, *(vi)* and *(vii)* are shown from the RTS game in 4.3. By showing how to build these primitives in relative isolation from each other, we are showing effectively a composable library that allows to create games where a player avatar may interact with obstacles in arbitrary rooms, and where some objects are shot by the players, others simply store useful properties of the map, others move and have some AI, and others create other units or manipulate the game world. Indeed, many games can be built recombining such components.

Since the games we present are simplified (it would be prohibitively difficult to build three large-scale games in this context) it is important to notice that these games represent just starting points that could be extended into full-blown games with more time and effort. An example of the extension process in action can be seen in [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.

The samples have been implemented in Casanova, F# under the RSD design patter, and idiomatic C# and XNA. We will focus on C# and XNA as our main point of comparison because they are used already by many of our target developers, under various different platforms: Windows PCs, Windows Phone 7, Android

and iOS smartphones (using Unity), the Xbox 360 and even the PlayStation Vita (using the PlayStation Suite SDK).

Notice that the samples shown below simply are some of the most representative snippets, that is the omitted entities (for example asteroids and projectiles) are not significantly more complex than those shown in the following.

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. The player aims the cannon and shoots the asteroids to prevent them from reaching the bottom of the screen.

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

  StarsSprite     : DrawableSprite
  ScoreText       : DrawableText

  Asteroids       : var<list<Asteroid>>
  Projectiles     : var<list<Projectile>>
  Cannon          : Cannon
  Score           : int }
  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>]
  rule Score(world,dt) =
    world.Score + [a | a <- state.Asteroids && a.Colliders.Length > 0].Length
  rule ScoreText.String(world,dt) = "Current score = " + world.Score.ToString()
```

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).

```
  { if is_key_down Keys.Left then return Some(true) else return None } => {
      state.Cannon.MoveLeft := true },
  { if is_key_down Keys.Right then return Some(true) else return None } => {
      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.

```
  { if is_key_down Keys.Space then return Some() else return None } => {
      state.Projectiles.Add
        { Sprite   = { Path   = "projectile.jpg"
                       Layer = world.Sprites }
          Position = vector(50.0<, 0.)
          Velocity = vector2(cos(state.CannonAngle),sin(state.CannonAngle))
          Colliders = [] }
      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 {
          Sprite   = { Path   = "asteroid.jpg"
                       Layer  = world.Sprites }
          Position = vector2(random(0.0<m>,100.0<m>),0.0<m>)
          Velocity = vector2(0.0<m/s>,random(5.0<m/s>,20.0<m/s>))
          Colliders = [] } }
```

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 = {
  …
  Colliders       : list<Projectile> }
  …
  rule Colliders(world,self,dt) =
    [x | x <- get_colliders world && distance(self.Position, x.Position) < 10.0f]
```

Listing 9 – Asteroids collision detection

Casanova has been studied with an important design goal: the ability to automatically optimize queries such as the one which computes the colliders of an entity; while we will not enter into the details of this process, suffice it to say that to quickly compute the above query Casanova automatically adds to the game world a spatial partitioning index of both asteroids and projectiles which then speeds up the collision query from $O(n^2)$ to $O(n \log n)$.

4.2 Game map and monsters with AI

The action/adventure game features a player-controlled character that moves between various rooms fighting monsters and drinking health-increasing potions. In the game we see how to handle: *(iv)* a game world comprised of *rooms and cells*, where each room is divided into cells; and *(v) monsters and their AI* who fight against the player. The game state contains, among other entities (such as the player) and sprite layers (to draw the game entities and the UI) the current room the player is in. A room is defined as a series of cells and a list of monsters; monsters are removed from a room whenever they are killed when fighting against the player (Listing 10).

```
type Room = {
    Cells             : list<list<Cell>>
    Monsters          : list<Monster> }
rule Monsters(world,self,dt) = [m | m <- self.Monsters && m.Health > 0]
```

Listing 10 - Room

Each cell contains, among various fields, a nullable reference to the room it leads to in case it is a portal, a list of neighboring cells, the room it leads to in case it were a door, a boolean value that determines if the player is in the cell, the list of monsters that are currently walking it, plus a sprite for drawing it (Listing 11).

```
type Cell = {
    Position      : Vector2<m>
    Sprite        : DrawableSprite
    Room          : ref<Room>
    HasPlayer     : bool
    Monsters      : list<ref<Monster>>
    Door          : Option<ref<Room>>
    Neighbours    : list<ref<Cell>> }
rule HasPlayer(world,self,dt) = world.Player.Position = self
rule Monsters(world, self, dt) =
  [m | m <- world.CurrentRoom.Monsters && m.Health > 0 && m.Position = self]
```

Listing 11 - Cell

Monsters are one the most important entities of the game. A monster contains fields to describe its position (the cell it's currently in), its target (the cell it's traveling to), the delta of its position between the current cell and the target cell, its sprite (for drawing it) and its current health. The monster rules update all of its fields; its health is updated when the player is in the same cell, the current cell is changed when the target cell is reached (and the target cell is set to null to stop the movement), and so on (Listing 12).

```
type Monster = {
    Position        : var<ref<Cell>>
    Sprite          : DrawableSprite
    Health          : int
    Damage          : var<int>
    MoveTarget      : var<ref<Option<Cell>>
    PositionDelta   : Vector2 }
member Movement(world,self,dt,on_arrived,on_moving) =
  match self.MoveTarget with
  | Some target ->
    if distance(self.Position.Position + self.PositionDelta, target.Position) < 0.1 then
      on_arrived target
    else on_moving()
  | None -> on_moving()
rule Health(world,self,dt) =
```

```
  if self.Position.HasPlayer && world.UpdateNow then
    self.Health - world.Player.Damage * random(1,4)
  else self.Health
rule Position(world,self,dt) =
  Movement(world,self,dt,id,fun () -> self.Position)
rule MoveTarget(world,self,dt) =
  Movement(world,self,dt,fun () -> None,fun () -> self.MoveTarget)
rule PositionDelta(world,self,dt) =
  Movement(world,self,dt,
 fun () -> self.PositionDelta + dt * normalize(target.Position - self.Position.Position),
 fun () -> self.PositionDelta)
rule Sprite.Position(world,self,dt) = self.Position.Position + self.PositionDelta
```

Listing 12 - Monsters

Monsters have a rudimentary AI (Listing 13) which chooses a destination cell for the monster to travel to, and which keeps doing so until the player is in the current room. This allows monsters to not weigh computationally when the player cannot interact with them, since he is in a different room.

```
  let monster_AI (monster : Monster) =
    repeat {
        if monster.Health > 0 then
          if world.UpdateNow && random(0, 20) < 1 && monster.MoveTarget = None then
            let target_cell = …
            do monster.MoveTarget := Some target_cell) }
    || { wait_condition (fun () -> world.CurrentRoom <> monster.Position.Room)
```

Listing 13 – Monster AI

Notice the use of the `(||)` operator which runs two scripts concurrently, that is until either of the scripts has completed.

4.3 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: *(vi) active entities*, such as planets, that represent complex components of the game scenario; and *(vii)* 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 (in the same spirit of the `UpdateNow` field in the Game of Life) allows the game battles to tick at fixed time intervals rather than at each tick of the game (Listing 14).

```
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 14 – 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 per-

centage 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 15).

```
type Planet = {
  Owner             : Player
  Armies            : var<int<Ship>>
  FractionalArmies  : float<Ship>
  AttackingFleets   : list<ref<Fleet>>
  ReinforcingFleets : list<ref<Fleet>>
  … }
rule Owner(world,self,dt) =
    if self.Armies <= 0 && self.AttackingFleets.Length > 0 then
      self.AttackingFleets[0].Owner
    else self.Owner
rule Armies(world,self,dt) =
  if self.Armies <= 0 then
    sum [a.Armies | a <- self.AttackingFleets && a.Owner = self.AtackingFleets[0].Owner]
  else
    let damages = sum[random(1,3) | f <- self.AttackingFleets] * state.TickBattles
    let reinforcements = sum[f.Armies | f <- self.ReinforcingFleets]
    self.Armies + int(self.FractionalArmies) - damages + reinforcements
rule FractionalArmies(world,self,dt) =
  self.FractionalArmies + (dt * self.Production) - floor(self.FractionalArmies)
rule AttackingFleets(world,self,dt) =
  [f : f <- state.Fleets && f.Target = self && f.Owner <> self.Owner && f.Arrived]
rule ReinforcingFleets(world,self,dt) =
  [f : f <- state.Fleets && f.Target = self && f.Owner = self.Owner && f.Arrived]
```

Listing 15 – 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 16).

```
  { if mouse_clicked_left() then
      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 16 – Planet selection

Similarly, when the user right clicks if there is an active selection some ships are sent (Listing 17).

```
  { if mouse_clicked_right() && world.SourcePlanet <> None then
      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 17 – Issuing orders

5 Final Assessment

Assessing the quality of a programming language for a given activity is a daunting task. Programming languages, much like natural languages, have a deep relation with the existing knowledge of the user. Indeed, there exist no metrics that make it clear that a language is good for a certain job, or even to compare two languages in a given context. 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, mainstream, general-purpose languages such as C# (plus libraries such as XNA) for real-time game development. We will discuss how Casanova

programs are overall much shorter than equivalent C# programs (measured excluding trivial constructs such as constructors or properties), and we will also discuss how the various snippets of code described in the sections above are shorter as well. Moreover, we will describe the amount of programming concepts that a developer must be fluent in to be able to program in Casanova, and we will discuss how this amount is rather smaller than the knowledge required for achieving the same in C#. We also include data from the same games implemented in F# with the Casanova library and according to the RSD pattern, to illustrate how RSD allows retaining many advantages of our framework, but in an existing language and its toolset.

The first comparisons that we make can be seen in Figure 1, and are 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.

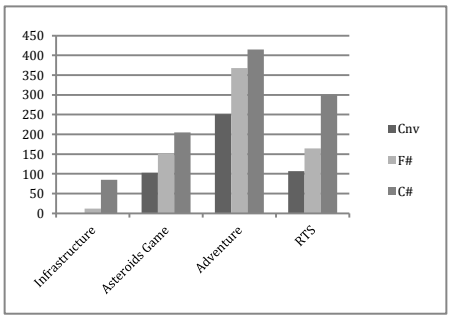

Figure 1

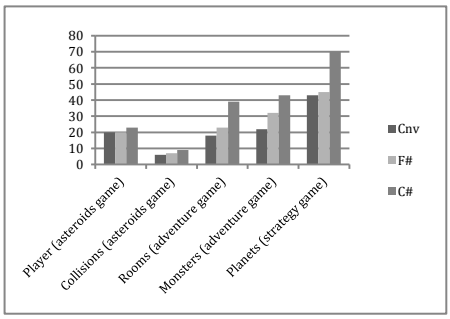

Figure 2

In Figure 2 we can see the comparison of the single snippets of game code that we have discussed in the previous section; we remark once again of the relevance of these snippets, since they can act as fundamental building blocks for a large number of games. 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#/RSD samples fare comparably to Casanova, both in terms of length and code complexity. Also, C# requires learning many more notions to be able to code a game in it: a minimum knowledge of object-orientation, events, and delegates is either necessary or required to avoid writing even more boilerplate code; Casanova is based on few primitives (records, rules and scripts) which have proven to be more than sufficient for capturing a great deal of patterns. Of course Casanova has some drawbacks. Performance is based on a higher-level framework (.Net) which features garbage collection, reflection, plus many other features that, while

not prohibitively costly, still mean that runtime speed will never be the same as that of using native languages. Also, the limitation that a rule may not write on any part of the state may cause some difficulty to imperative programmers who are used to think about code in terms of actions and assignments rather than atomic entities and fields which update strictly themselves. Note that the current (prototypical) implementation is based on reflection to traverse the state, and so it is slower than the final version that will be output by a compiler. Fortunately, Casanova does not force on the developer a particular model of the world or its entities, and so its limitations are all focused on linguistic or runtime issues rather than intrinsic expressivity issues, such as the inability to represent the player or another entity in a certain way. 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. The game features thousands of simultaneous entities, real-time 3D graphics with special effects, and even real-time multiplayer via LAN and Internet of up to 8 players. The game clearly shows the feasibility of Casanova, especially the power of rules and scripts, when used on a larger scale.

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 some "fringe" game developers, such as independent game developers, serious game developers, and even research game developers, who traditionally have neither the budget nor the manpower to tackle some of the challenges associated with making a modern game. Casanova is a step in this direction: by studying the art and craft of game development we are building a

framework and a language that simplify many tasks and allow game developers to put more effort on AI, gameplay, shaders and other important tasks such as procedural generation rather than on the "nuts-and-bolts" of putting a game together and optimizing it.

In conclusion, Casanova allows a developer to build shorter games when compared to traditional languages; it also requires very little programming concepts to be used proficiently; the scripting system of Casanova (and its F# rendition) removes the need for hand-crafting complex state-machines, using threaded systems or building event-based mechanisms. 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.

8 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 Proceedings of the 13th conference on Advances in Computer Games, ACG 13, Tilburg, 2011, Springer. In : 13th Internation Conference Advances in Computer Games (ACG), Tilburg, Netherlands (2011)
7. Maggiore, G., Bugliesi, M., Orsini, R.: Casanova Papers. In: Casanova project page. (Accessed 2011) Available at: http://casanova.codeplex.com/wikipage?title=Papers
8. DeLoura, M.: The Engine Survey. In: Gamasutra. (Accessed 2009) Available at: http://www.gamasutra.com/blogs/MarkDeLoura/20090316/903/The_Engine_Survey_Technology_Results.php
9. Ampatzoglou, A., Chatzigeorgiou, A.: Evaluation of object-oriented design patterns in game development. In : Journal of Information and Software Technology, MA, USA, vol. 49 (2007)
10. Folmer, E.: Component based game development: a solution to escalating costs and expanding deadlines? In : Proceedings of the 10th international conference on Component-based software engineering, CBSE, Berlin, Heidelberg, p.66– 73 (2007)
11. Conal, E., Hudak, P.: Functional reactive animation. In : International Conference on Functional Programming (ICFP), p.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, p.31–42 (2007)
13. Maggiore, G.: Casanova project page. In : http://casanova.codeplex.com/ (2011)
14. Maggiore, G.: Galaxy Wars Project Page. In : http://vsteam2010.codeplex.com, http://galaxywars.vsteam.org (2010)
15. Richard Zhao, D.: Generating Believable Virtual Characters Using Behaviour Capture and Hidden Markov Models. In : 13th Internation Conference Advances in Computer Games (ACG), Tilburg, Netherlands (2011)
16. Garcia-molina, H., Ullman, J., Widom, J.: Database System Implementation. (1999)