The Rule-Script-Draw Design Pattern for Making Games

Giuseppe Maggiore, Renzo Orsini, Michele Bugliesi

Università Ca' Foscari Venezia
DAIS - Computer Science
{maggiore,orsini,bugliesi}@dais.unive.it

Abstract. In this paper we present the Rule-Script-Draw design pattern around which the Casanova language is structured. This pattern analyzes the common structure of a game and how these structures interact with each other.

Keywords: Game development, Casanova, design patterns, languages, functional programming, F#

1 The Rule-Script-Draw Design Pattern

Casanova is a programming language that is being designed in order to facilitate creating games. Casanova is built around an idea of games and their essential components that is rather novel, since there is no disciplined work in the literature that tries to identify the unique components that make up a game from the software engineering standpoint.

After a careful analysis of the structure of different games, we have realized that games are defined in terms of three main components, and their interaction. On one hand there is the logic of the game, which at every tick of the logic simulation modifies the data structures that describe the world so that they reflect the world as updated after a certain amount of time; on the other hand there is the drawing of the simulation, which takes the current state of the world as described by its data structures and performs a series of drawing operations. Game development frameworks such as DirectX UT, XNA, and many others are all rooted in a simpler model of games based on the main loop, which offers customizable (or virtual, or scriptable) functions that are then invoked by the framework at the fixed time intervals and that implement the game logic and rendering. This architecture means that the only views the developer has of the state are fragmented with respect to time, meaning that any code operation must be performed and completed within a single tick of the simulation (usually taking no less than $1/30^{th}$ of a second). Many game operations, from animations to timers to complex sequential behaviors are ill-suited to this kind of representation, while many others fit perfectly. Modern game engines, such as Unity (and many, many others) have

adopted scripting engines and coroutine mechanisms that make it easier to integrate long-running operations in the game loop.

Observing these aspects, we have come up with a description of a game in terms of four main components:

- The description of the world and its entities as a series of data structures
- The description of how an entity updates itself and its fields (by reading, but not writing the rest of the world) as a series of **rules** attached to each entity
- The description of a series of imperative **scripts** that, similarly to threads, run in sequence, parallelism and concurrency with each other and which modify the variable parts of the state
- The description of all the **drawable** fields of each entity which are rendered automatically after each tick of the update function

A game defined in terms of these four aspects follows the **rule-script-draw (RSD)** design pattern, which we believe is already followed unwittingly by many games and around which better (and explicit) abstractions should be built.

A single frame thus performs the following operations:

- We start with $world_t$, the world description at time t
- We apply all the rules of the game, executed in parallel with each other (that is, with no interference, and possibly also in a multithreaded fashion so as to speed up their evaluation)
- We then run a step of each active script of the game, sequentially since scripts may have side effects and thus each of them modifies the state in a way that may be significant to the other scripts
- Finally we draw all the drawable entities contained in the game world, and we repeat the process

2 The Casanova language

The Casanova language is similar to the languages of the ML family (F# in particular, with a few aspects such as list comprehensions inspired from the elegant Haskell syntax) and it is built around the idea of defining a game where the logic and drawing of the game are declarative, plus a series of imperative processes (scripts) that take care of the user input and other sequential operations. We now describe Casanova by showing how to build an application in it. 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. A matrix of cells is changed according to the following rules: *(i)* a cell with less than two alive neighbors dies because of under-population; *(ii)* a cell with more than three neighbors dies because of over-population; *(iii)* any other cell remains the same.

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 (such as transforms, alpha blending, shader, etc.).

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

Each cell 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 is updated to reflect its current value. 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. A cell also contains a sprite, which is drawn automatically by the framework after each tick of the simulation.

```
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
```

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. Notice that when assigning the fields of a rendering entity or layer, we just specify those parameters we are interested in; all other parameters are assigned default values.

```
let initial_world =
  let make_cell i j =
    { NearCells = []
      Value     = random(0, 4) / 3
      Sprite    = { Path     = "WhitePixel.bmp"
                    Position = vector2(i,j)
                    Layer    = sprites }

  let sprites = { Transform   = Matrix.Identity; AlphaBlend  = false }
  let world_aux =
    { Sprites   = sprites
      Cells     =
        [ for i = 1 to 100 do yield [ for j = 1 to 100 do yield make_cell i j } } ] ]
      UpdateNow = false }
  … // setup neighbors
```

```
  world_aux
```

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.

After the definition of the game entities, rules and drawable fields, come the definition of the game scripts: the main script for the imperative logic of the game, and the input script for the input management of the game. The main script 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:

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

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 Separating Logic and Rendering with RSD

A very important concern when working on a larger project is that of separating game logic and game rendering, so that different people in the team may work on each without too much interference. The RSD pattern supports such a division that is the various entities of the game may be separated in visual and logical subentities, each of which is processed separately with its own rules, and which are connected by a further set of rules which take information from the logical state of the entities and puts it into the visual state for drawing.

Let us consider a simple example of an entity which represents the ship that a player moves with some input commands and which is also rendered in the form of a 2D sprite.

A first, naïve, implementation might simply put everything into the same datastructure, both the logical values of the ship (which in this case model its rather simple physics) and the drawable sprite of the ship. A series of rules both update the data of the logical values and "translate" the logical data into values for the drawable sprite so that it may be rendered correctly:

```
type Ship = {
  Position      : Vector2<m>
  Velocity      : float<m/s>
  Acceleration  : var<float<m/s^2>>
  Rotation      : var<float<rad>>
  Integrity     : float
  Sprite        : DrawableSprite
}
rule Position(world,self,dt) = self.Position + Vector2(cos self.Rotation, sin
self.Rotation) * self.Velocity * dt
rule Velocity(world,self,dt) = self.Velocity + self.Acceleration * dt
rule Acceleration(world,self,dt) = 0.0<m/s^2>
```

```
rule Sprite.Position(world,self,dt) = self.Position
rule Sprite.Rotation(world,self,dt) = self.Rotation
rule Sprite.Color(world,self,dt) = Color.Interpolate(Color.Red, Color.White,
self.Integrity)
```

As needed, it is possible to divide the above definition in three parts: *(i)* the logic of the asteroid, which is now fully-contained; *(ii)* the drawable parts of the asteroid, separated from the logic; *(iii)* an entity that contains both the logic and the drawing of the asteroid and which has rules that transfer information from the logic and into the rendering:

```
type ShipLogic = {
  Position      : Vector2<m>
  Velocity      : float<m/s>
  Acceleration  : var<float<m/s^2>>
  Rotation      : var<float<rad>>
  Integrity     : float
}
rule Position(world,self,dt) = self.Position + Vector2(cos self.Rotation, sin
self.Rotation) * self.Velocity * dt
rule Velocity(world,self,dt) = self.Velocity + self.Acceleration * dt
rule Acceleration(world,self,dt) = 0.0<m/s^2>

type ShipDrawing = {
  Sprite  : DrawableSprite
}

type Ship = {
  Logic   : ShipLogic
  Drawing : ShipDrawing
}
rule Drawing.Sprite.Position(world,self,dt) = self.Logic.Position
rule Drawing.Sprite.Rotation(world,self,dt) = self.Logic.Rotation
rule Drawing.Sprite.Color(world,self,dt) = Color.Interpolate(Color.Red, Color.White,
self.Logic.Integrity)
```

Even though in the above sample the drawing of the ship is trivial, and so an entire data-structure is not strictly needed, this approach is useful, extensible and maintainable from the point of view of the graphics effects developer who may have the need to add to the `ShipDrawing` data structure further fields and rules which represent the current state of the visual simulation for the entity.

As an example of this process, in the Galaxy Wars strategy game, each logical entity contains its drawable data, which is then updated in a separated context: fleets of ships in particular need to store their position and their current damage points, but they are represented with a squadron of many ships; these ships do not have any role in the logical simulation, but they greatly increase the feeling of battle. The ships, their flocking algorithm and all their information reside in the drawable portions of the data.

Furthermore, Casanova allows a developer to define a data contract (essentially a statically resolved abstract class which may then be inherited by an entity. We may then define a contract for the logic of the ship which contains all the fields and rules of the ship logic:

```
contract ShipLogic = {
  Position      : Vector2<m>
  Velocity      : float<m/s>
  Acceleration  : var<float<m/s^2>>
  Rotation      : var<float<rad>>
```

```
  Integrity     : float
}
rule Position(world,self,dt) = self.Position + Vector2(cos self.Rotation, sin
self.Rotation) * self.Velocity * dt
rule Velocity(world,self,dt) = self.Velocity + self.Acceleration * dt
rule Acceleration(world,self,dt) = 0.0<m/s^2>
```

We then define another contract, which is parameterized over the type that it will assume (which it requires to be a `ShipLogic`) that contains all the rendering fields and rules:

```
contract ShipDrawing<'self : ShipLogic> = {
  Sprite  : DrawableSprite
}
rule Sprite.Position(world,self:'self,dt) = self.Position
rule Sprite.Rotation(world,self:'self,dt) = self.Rotation
rule Sprite.Color(world,self:'self,dt) = Color.Interpolate(Color.Red, Color.White,
self.Logic.Integrity)
```

Finally, we state that the ship is both a `ShipLogic` and a `ShipDrawing`:

```
type Ship = ShipLogic, ShipDrawing<ShipLogic>
```

4 Conclusions

Game development is a large and important aspect of modern culture; games are used for entertainment, education, training and more, and their impact on society is very large. 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 tradition; it also requires very little programming concepts even though some advanced uses of collections may require learning higher-order functions; 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 (Galaxy Wars: http://galaxywars.vsteam.org), and we are certain that with further work the benefits of this approach will become much more apparent.