Making Games with Casanova

Giuseppe Maggiore, Alvise Spanò, Renzo Orsini, Michele Bugliesi, Enrico Steffinlongo
Università Ca' Foscari Venezia
DAIS - Computer Science
{maggiore,spano,orsini,bugliesi,esteffin}@dais.unive.it

ABSTRACT
In this paper we present the Casanova language and framework for making games. We show how Casanova is suitable for making games of all genres and we detail how to build a series of simple but very different games: the Game of Life, a shooter game, a strategy game and a role-playing game. We discuss the difference between the Casanova language and the Casanova framework, and we outline the many extensions that we are studying for addition to this work.

Categories and Subject Descriptors
D.2.2 [**Software Engineering**]: Design Tools and Techniques –*Software libraries*

D.3.3 [**Programming Languages**]: Language Constructs and Features – *abstract data types, polymorphism, control structures*

H.5 [**Information Interfaces and Presentation**]: Multimedia Information Systems – *Artificial, augmented and virtual realities*

General Terms
Performance, Experimentation, Human Factors, Languages.

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

1. INTRODUCTION
Games are a huge business [1] and a very large aspect of modern popular culture.

Independent games, the need for fast prototyping gameplay mechanics [2] and the low budget available for making serious games [3] (when compared with the budget of blockbuster games) has created substantial interest in research on principled design techniques and on cost-effective development technologies for game architectures. Our present endeavor makes a step along these directions.

Making games is an extremely complex business. Games are large pieces of software with many heterogeneous requirements, the two crucial being high quality and high performance [4].

High quality in games is comprised by two main factors: visual quality and simulation quality. Visual quality in games has made huge leaps forward, and many researchers continuously push the boundaries of real-time rendering towards photorealism. Simulation quality, on the other hand, is often lacking in modern games; game entities often react to the player with little intelligence, input controllers are used in simplistic ways and the logic of game levels is more often than not completely linear. Building a high-quality simulation is very complex in terms of development effort and also results in computationally expensive code. To make matters worse, gameplay and many other aspects of the game are modified (and often even rebuilt from scratch) many times during the course of development. For this reason game architectures require a lot of flexibility.

To manage all this complexity, game developers use a variety of strategies. Object-oriented architectures, external scripting languages, components, reactive programming, etc. have all been used with some degree of success for this purpose [5] [6] [7] [8].

In this paper we will present the Casanova language and framework. Casanova is the result of a search, started with [9,4], for a general-purpose methodology that makes game development easier and speedier, by integrating many of the benefits of the above-mentioned techniques.

In the remainder of the paper we show the Casanova language in action. We begin with an informal description of Casanova and its underlying computational model for games and we make a comparison with existing approaches in section 2. We give a description of the Casanova language in section 3. We discuss how to make actual games with Casanova (by giving detailed examples) in section 4. In section 5 we describe the many extensions that we are planning on adding to Casanova; these extensions touch areas like networking, graphics, AI, and many others and we believe these to be extremely important in the overall direction of the project.

2. WHAT IS CASANOVA?
Casanova is a programming language that offers a mixed declarative and procedural style of programming which has been designed in order to facilitate game development. The final goal of the language is to require from the developer to code only and exclusively those aspects of the game code specific to the game being developed, rather than writing lots of boilerplate code. The language aims for simplicity and expressive power, but it is also designed in order to allow powerful automated optimizations. The language ensures consistency in the update of the game world and it also offers integrated support for coroutines, a mechanism often used [10] in games.

Casanova comes in two flavors: a library (framework) for making games and the full-fledged language. The language actually compiles into the library, but until the language is complete with all its supporting tools and it is well integrated into an existing IDE (i.e. Visual Studio, Eclipse or MonoDevelop) to make our work truly useful to game developers we have built a .Net library that incorporates various features of Casanova.

Casanova solves the following problems encountered when making a game: *(i)* updating the world *consistently*, that is the world is never stored partially updated; *(ii)* traversing the world *fully*, that is no portion of the world is left out of an update; *(iii)* expressing *queries* on the many lists of entities of the world; *(iv)* supporting *imperative processes* that operate on the game state.

Each updateable entity of the game state defines the computation that transforms its fields from the current time step to the next. These computations are called *rules*. Each rule is applied exactly once on each entity it upgrades, and at the same time with all other rules to avoid any temporal interference. This technique allows us to avoid many bugs that derive from the state being updated in place (for an example, see section 4).

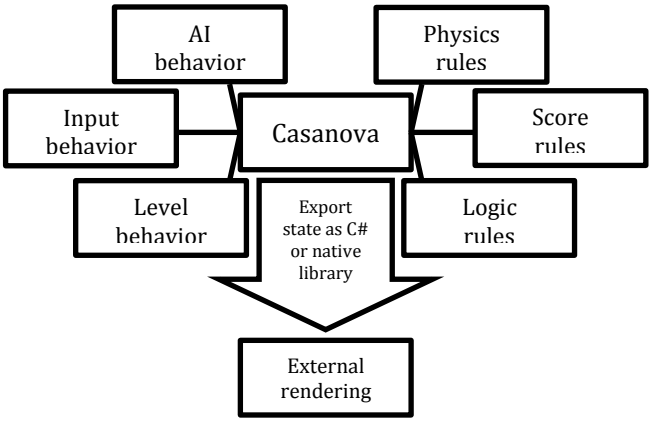
A game features many different entities of the same type. Enemies, units, rooms, and projectiles: the more entities there are the more engaging and vibrant the game world becomes. Casanova simplifies working with collections as much as possible by supporting declarative queries (in the style of databases) over lists of entities. Queries are optimized by building indices (spatial partitioning is a special case of this optimization) according to the same rules that govern optimization of database queries.

While many of the aspects of a game can be successfully described declaratively, other aspects (such as AI, input, etc.) are more imperative in nature; the behavior of an enemy character is best described as a series of orders, to be performed concurrently w.r.t the regular update routine of the game world. To support these aspects of a game, Casanova supports *behaviors* (implemented through coroutines) [11] for creating scripts, managing input, implementing the level activators, etc.

2.1 Architecture of a Casanova Game

A Casanova game (either built with the Casanova framework or the Casanova compiler) is exported as a library. This library implements the data-types that describe the game world, and the update function that computes a time-step of the world data-types. After each update an external rendering system draws the current state of the world to the screen.

Casanova does not (yet) implement a rendering system. For this reason the architecture of a Casanova game can be described with the figure below:



2.2 Comparison with existing approaches

The two most common game engine architectures found in today's commercial games 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 game engine [6].

A component-based system defines 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 [8].

These two more traditional approaches both suffer from a noticeable shortcoming: they focus exclusively on representing single entities and their update operations in a dynamic, even composable way. 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. Also, all games feature behaviors that take longer than a single tick; these behaviors are hard to express inside the various entities, which often end up storing explicit program counters to resume the current behavior at each tick. Moreover, 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 [7]) is studied in the context of functional languages. FRP 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 it addresses 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 query engine for

defining a game. This query engine is optimized in order to make it simple to express efficient aggregations and Cartesian products, two very common operators in games. On the other hand, 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 three issues in mind: with Casanova, the integration of the interactions between entities and long-running behaviors is seamless, and the resulting game world is always consistent.

3. 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 according to a precise model.

List comprehensions have a syntax that is similar to the set notation commonly used in mathematics; a list comprehension returns the values of expression evaluated for each combination of values of a series of range variables, each of which iterates all the values in a collection. Those values of the range variables that fail certain predicates are discarded:

```
[ expr(x1,x2,...) | x1<-l1 && x2<-l2 && ... pred1(x1) && ... ]
```

We begin by defining the datatypes that describe the game world in a record called `GameState`. The game state, like any other record in the language, is a series of fields that can either be simple variables, other records or collections of values. Each field may have at most one rule attached to it; a rule computes the value that the field will assume at the next iteration of the game loop. Rules are defined according to the following syntax:

```
rule FieldName : FieldType = rule_function
```

`rule_function` is a pure (that is, side-effect free) function that takes as input the state of the world (which is called `GameState`), the entity E and the time between this update and the last one and which returns the new value of the field. The type of the `rule_function` is:

```
GameState x E x float -> FieldType
```

Rules are the main work-horses of Casanova games; for example, a rule may update the position of an entity by incrementing it with its velocity:

```
rule Position : Vector2 =
  fun (s,e,dt) -> e.Position + dt * e.Velocity
```

or a rule may cull all the elements of a collection that are dead or disabled:

```
rule Items : Table SomeEntity =
  fun (s,e,dt) -> [x | x <- e.Items && e.Disabled = false]
```

Conceptually, all rules are evaluated simultaneously, that is all rules are applied on the same game state and they produce the new game state without interferences. Casanova does not allow the user to define an update function outside rules that is all that happens during a tick of a game must be specified with rules.

After defining the game world datatypes and their rules we define the initial state, which is the state of the world when the game is launched and before the first update.

Rules are high-level, expressive constructs and being declarative they allow for many optimizations (see 4.5 for more details). As such, all that can be written in terms of Casanova rules should be. This said we recognize that rules sometimes can be awkward to use, and a more imperative, straightforward approach may be needed. To address this shortcoming so we have built an additional scripting system to specify imperative *behaviors* with coroutines, which smoothly integrate with rules. We use the same monadic system of the F# language [13] to build the system of coroutines which to define the main imperative process interleaved with the update loop of the game [9]. Coroutines are sequential programs that suspend themselves for the rest of the update cycle (also known as "tick") through the `yield` statement. Coroutines invoke each other with the `do!` and `let!` monadic operators [13]; the former does not expect a returning value, while the latter does. When the invoked coroutine suspends itself with a `yield`, then the caller suspends as well. It is worthy of notice that our system is similar to the scripting systems based on coroutines that many games use, even though the degree of integration of our coroutine system with the rest of the game engine is higher when compared with that of commonly used mechanisms which typically "attach" to the main engine an external scripting language with ungainly binding mechanisms [10].

Rules, consistency, and the behavior system are also implemented as an F# library, while the optimizations are (as of the time of writing) only available through the Casanova language.

4. MAKING GAMES WITH CASANOVA

Casanova aims towards making game development easier by integrating patterns that offer various aids to a game developer.

The project is mostly aimed at independent game developers, smaller studios and prototype developers; Casanova might also benefit serious game developers, and in general all those studios that do not have the same development resources typical of major commercial projects. In time the language will generate an appropriate renderer for the game state, it will integrate ready-made AI modules and it will support synchronization of the game state across a network for multiplayer. In short, Casanova aims at taking over all areas of game development, starting from the definition of the game logic and then spiraling outward towards all other areas.

We are aware that a project that aims at making games *in general* is tackling a very broad problem. For this reason in this paper we present a series of game stubs that implement the core of various very different games, in order to show the flexibility of the system when defining games belonging to different genres. We show how to build: *(i)* the simulation known as the *Game Of Life*, which we have found to be a good introduction to Casanova; *(ii)* a shooter game where a starship must destroy a series of falling asteroids; *(iii)* an action-RPG game where the player moves from a room to another fighting enemies and looking for health potions; *(iv)* an RTS game where the player must conquer various planets by moving his ships from one planet to another.

We will show the various samples in detail, but for space constraints we have greatly simplified the games to the point that their gameplay is quite diminished. The samples (implemented in both Casanova and F# [14] with the Casanova framework for the game logic and XNA framework [15] for rendering) can be downloaded from [16]. Our goal is to show that these Casanova games could be extended into real games, even though with extensive additions; an example of this process can be seen in [17], a strategy game that we are building as an ongoing study of how to create non-trivial games with Casanova.

4.1 GAME OF LIFE

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:

```
type GameState = {
  Cells       : Table Table 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. The value of the cell is updated every time an update is triggered, by summing the value of the neighbors and applying the rules mentioned above:

```
type Cell = {
  NearCells   : Ref Table Cell;
  rule Value  : int =
    fun (state, 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 }
```

The initial state of the game creates the matrix of cells and initializes their neighbors. Each internal cell has exactly eight neighbors:

```
let state =
  let state = {
    Cells =
     [ for i = 1 to 100 do
        yield [ for i = 1 to 100 do
                  yield { NearCells = [];
                          Value = random(0,4) / 3
                          UpdateNow = false } ] ] }
  // set up each cell neighbors
  in state
```

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 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 state =
  let rec behavior() = {
      do! wait 1.0
      do! state.UpdateNow := true
      do! yield
      do! state.UpdateNow := false
      do! behavior() }
  behavior()
```

This way the game of life will run at exactly one step per second, no matter the framerate of the simulation.

4.2 ASTEROID SHOOTER

The asteroid shooter game is a simple shooter game where asteroids fall from the top of the screen towards the bottom. The player aims a cannon and shoots the asteroids to prevent them from reaching the bottom of the screen.

The game state contains: *(i)* a table of asteroids, which rule produces the collection of updated asteroids except those that reach the bottom of the screen or that hit a projectile (which are thusly removed); *(ii)* a table of projectiles, which are removed when they reach the top of the screen or when they hit an asteroid; *(iii)* the current direction the cannon is aiming at, which is updated via the user input and *(iv)* the score counters:

```
type GameState = {
  rule Asteroids      : Table Asteroid =
    fun (state, self, dt) ->
      [a | a <- state.Asteroids && a.Colliders.Length = 0 &&
           a.Y < 100.0f]
  rule Projectiles    : Table Projectile
    fun (state, self, dt) ->
      [p | p <- state.Projectiles && p.Colliders.Length = 0 &&
           p.P.Y > 0.0f]
  rule CannonAngle    : float32 =
    fun (state, self, dt) ->
      asteroids_state.CannonAngle +
      if is_key_down Keys.Left then -dt
      elif is_key_down Keys.Right then dt
      else 0.0f
  DestroyedAsteroids  : Var int
  MissedAsteroids     : Var int }
```

Asteroids are updated by increasing their `Y` coordinate according to their velocity; an asteroid also stores the list of projectiles that are currently colliding with it:

```
type Asteroid = {
  X               : float32
  rule Y          : float32 =
    fun (state, self, dt) -> self.Y + dt * self.VelY
  VelY            : float32
  rule Colliders  : Ref Table Projectile
    fun (state, self, dt) ->
      [p | p <- state.Projectiles &&
distance(Vector2(self.X,self.Y), p.P) < 10.0f] }
```

A projectile is very similar to an asteroid, but its velocity is a 2D vector rather than a vertical direction, and it stores the list of asteroids that are currently colliding with it:

```
type Projectile = {
  rule P          : Vector2
    fun (state, self, dt) -> self.P + dt * self.V
  V               : Vector2
  rule Colliders  : Ref Table Asteroid
    fun (state, self, dt) ->
      [a | a <- state.Asteroids && distance(Vector2(a.X,a.Y),
           self.P) < 10.0f] }
```

The initial state contains no asteroids and no projectiles, the cannon points upwards and the score counters are both zeroed. We do not show the source for the initial state, as it is trivial.

The main script takes care of generating random asteroids and reading the user input to shoot projectiles. Asteroids

are generated by waiting a random interval of 1 to 3 seconds, then creating an asteroid with a random x coordinate and a random velocity and then repeating the process:

```
let main state =
  let rec spawn_asteroids() =
  { do! wait (random (1.0,3.0))
    add state.Asteroids
        {
          X = random_float(0.,100.)
          Y = 0.0f
          VelY = random_float(5.0,20.0)
          Colliders = [] }
    do! spawn_asteroids () }
```

Projectiles are generated by waiting for the user to press the space button; when space is pressed, then the projectiles is generated with a velocity that corresponds to the current aim of the cannon. To avoid generating one projectile every frame, we wait one-fifth of a second every time a projectile is generated; this way even if the user holds the space button for a long period of time the number of projectiles shot every second will be constant (namely, 5):

```
  let rec shoot_projectiles() = {
    do! yield
    if is_key_down Keys.Space then
      add state.Projectiles
          {
            P = Vector(50., 0.)
            V =
              let x = cos(state.CannonAngle)
              let y = sin(state.CannonAngle)
              Vector2(x,y) * 10.0f
            Colliders = []
          }
      do! wait 0.2
    do! shoot_projectiles() }
```

After defining the internal script, the main script launches in parallel both the spawning of asteroids and the shooting of projectiles:

```
  shoot_projectiles() && spawn_asteroids()
```

4.3 ROLE-PLAYING-GAME

The RPG features a player-controlled character who moves between various rooms fighting monsters and drinking health-increasing potions. The state of the game contains the player, the list of rooms (which are all references, that is they are not updated), the current room (the only one which will be updated) and a boolean variable which, like in the *Game of Life*, inhibits certain rules from happening once per frame and thus updating the state too quickly for the user to see:

```
type GameState = {
  Player        : Player
  Rooms         : Ref Table Room
  CurrentRoom   : Var Room
  UpdateNow     : Var bool }
```

A room contains a matrix of cells, and stores the list of monsters of all its cells. Each cell has a position, and stores a list of monsters and potions, and it tracks the player and its neighboring cells; a cell may also be a door that is it may lead to another room:

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

type Cell = {
  Position       : Vector2
  rule HasPlayer : bool =
    fun (state, self, dt) -> state.Player.Position = self
  rule Monsters  : Table Ref Monster =
    fun (state, self, dt) ->
      [m | m <- state.CurrentRoom.Monsters &&
          m.Health > 0 && m.Position = self]
  rule Potions   : Table unit =
    fun (state, self, dt) ->
      if self.HasPlayer then []
      else self.Potions
  Door               : Var Ref Option Room
  Neighbours         : Ref Table Cell }
```

The player has a position (the cell they currently are on), a health, the damage he inflict during battle, a movement target and a movement delta. The health automatically decreases because of battles against monsters in the same cell and it increases thanks to potions. Monsters are almost identical to the player and so we do not show them:

```
type Player = {
  Position        : Var Ref Cell
  rule Health     : int =
    fun (state, self, dt) ->
      self.Health –
      if state.UpdateNow then
        sum [m.Damage * random(1,4) |
          m <- self.Position.Monsters && m.Health > 0] +
        sum [random(10,40) | p <- self.Position.Potions]
      else 0.0f
  MaxHealth       : Var int
  Damage          : Var int
  Target          : Var Ref Option Cell
  MovementDelta   : Var Vector2 }

type Monster = { (* very similar to Player *) }
```

The initial state of the game initializes the game rooms and their monsters and potions (either procedurally or by loading them from a file) and it initializes the player in a cell of the first room.

The main script performs three important functions: *(i)* ticks the `UpdateNow` field once every second; *(ii)* handles the user input; *(iii)* handles the player and monsters movement. We do not show *(i)* because it is identical to its counterpart in the *Game of Life*.

```
let main =
let rec tick () = { ... // identical to the Game of Life }
```

Monsters move by going from their current cell to their target if they have one. A monster ends it behavior if it dies or if the current room changes:

```
let rec monster_movement (monster : Monster) = {
    do! yield
    if monster.Health > 0 &&
       monster.Room = state.CurrentRoom &&
       state.UpdateNow then
      match monster.Target with
      | None -> do! monster_movement monster
      | Some target ->
        let source = monster.Position.Position
        let destination = target.Position
        let delta = destination - source
        let move (dt : float) = {
            do monster.MovementDelta :=
                lerp(Vector2.Zero, delta, dt * step) }
        do! wait_doing move 1.0
        do monster.Position := target
        do monster.Target := None
        do monster.MovementDelta := Vector2.Zero
        do! monster_movement monster }
```

We omit, for brevity, the monster “AI” which randomly select a neighboring cell as the monster target:

```
let rec monster_AI (monster : Monster) = { ... // omitted }
```

The player movement is very similar to the monster movement (to the point that we omit part of the player movement routine), with an important difference: after movement, the player checks if the current room has changed. If it has, then a movement and AI script is launched for each monster of the new room, that is those monsters are activated:

```
let initialize_current_room () = {
    for m in state.CurrentRoom.Value.Monsters do
      do! add_script (monster_movement m)
      do! add_script (monster_AI m) }

let rec player_movement () = {
    ... // same movement code of the monster script
    match state.Player.Position.Door with
    | None -> ()
    | Some room ->
      do state.CurrentRoom := room
      do state.Player.Position :=
        room.Cells[int destination.X][int destination.Y]
      do! initialize_current_room ()
    do! player_movement () }
```

The player input simply reads the user input and resets the player target cell:

```
let rec input () = {
    if state.UpdateNow then
      let pos = state.Player.Position.Position
      if is_key_down Down && pos.Y < 9. then
        let target = state.CurrentRoom.Cells[int pos.X][int
(pos.Y + 1.)]
        do state.Player.Target := Some target
      elif is_key_down Up && pos.Y > 0. then
        ... // other cases as above
    do! yield
    do! input () }
```

The actual body of the main script initializes the current room and launches the ticking function, the input script and the player movement script all in parallel:

```
{ do! initialize_current_room()
  do! tick() && (input() && player_movement ()) }
```

4.4 STRATEGY GAME

The strategy game features a series of planets that produce ships, which can then be sent to conquer other planets. We have expanded upon this prototype in order to see how far Casanova can go when building a complex, articulated game. The result [16] shows that indeed Casanova games can be expanded upon and can achieve much greater complexity than that shown in the paper.

The state of the game contains a list of planets and a list of fleets. Fleets are culled from the state when they reach their destination and finish fighting. Similarly to the *Game of Life*, we use a variable to inhibit the application of some rules (those that implement the battles) so that they do not change the state too quickly:

```
type RTSState = {
  Planets       : Table Planet
  rule Fleets   : Table Fleet =
    fun (state, self, dt) ->
      [f | f <- self.Fleets && f.Alive &&
           (not f.Arrived || f.Fighting)]
  TickBattles  : Var bool }
```

A planet contains an owner, its position, the current number of armies it hosts, the progress in building the next army, the production rate and the list of enemy fleets inbound to the planet. The owner of the planet and the current armies are modified according to the result of battle against the attacking fleets; when the planet armies are all destroyed, then the planet is owned by the owner of the first attacking fleet:

```
type Planet = {
  Position                    : Vector2
  rule Owner                  : Player =
    fun (state, self, dt) ->
      if self.Armies <= 0 &&
         self.AttackingFleets.Length > 0 then
        self.AttackingFleets.[0].Owner
      else self.Owner
  rule Armies                 : int =
    fun (state, self, dt) ->
      if self.Armies <= 0 then
        sum [a.Armies | a <- self.AttackingFleets &&
                        a.Owner = self.AttackingFleets[0].Owner]
      else
        let damages =
          if state.TickBattles then
            [random(1,3) | f <- self.AttackingFleets]
          else []
        (if self.FractionalArmies > 1.0 then
           self.Armies + 1
         else self.Armies) - sum damages
  rule FractionalArmies       : float =
    fun (state, self, dt) ->
      if self.FractionalArmies < 1.0 then
        self.FractionalArmies + (dt * Production)
      else self.FractionalArmies - 1.0
  Production                  : float
  rule AttackingFleets        : Ref Table Fleet =
    fun (state, self, dt) ->
      [f | f <- state.Fleets && f.Target = self &&
           f.Owner <> self.Owner && f.Arrived] }
```

Fleets contain a position, an owner, a velocity, a target, a number of armies and various indicators that determine the current state of the ship: whether it is fighting, traveling and still alive. The ship updates its position according to its velocity and its armies according to battle:

```
type Fleet = {
  rule Position   : Vector2 =
    fun (state, self, dt) ->
      self.Position + self.Velocity * dt
  Owner           : Player
  Target          : Ref Planet
  rule Arrived    : bool =
    fun (state, self, dt) ->
      distance(self.Position, self.Target.Position) < 10.0
  rule Velocity   : Vector2 =
    fun (state, self, dt) ->
      if self.Arrived then Vector2.Zero
      else self.Velocity
  rule Armies     : int =
    fun (state, self, dt) ->
      if self.Fighting && state.TickBattles then
        self.Armies - random(1,3)
      else
        self.Armies
  rule Fighting   : bool =
    fun (state, self, dt) ->
      self.Arrived && self.Target.Owner <> self.Owner &&
      self.Alive
  rule Alive      : bool =
    fun (state, self, dt) -> self.Armies > 0 }
```

The game only features two players: a human and an AI:

```
type Owner = Human | AI
```

The initial state of the game creates a set of planets that can either be preset or loaded from a file; at the beginning of the game there is no fleet active.

The main script of the game is responsible for creating the various fleets according to user input and a rudimentary AI:

```
let main =
  let mk_fleet source target owner = {
      let num_armies = max 1 (source.Armies / 2)
      source.Armies := source.Armies - num_armies
```

```
      let new_fleet =
        { Position    = source.Position
          Velocity    = normalize(target.Position –
                           source.Position) * 0.5
          Armies      = num_armies
          Owner       = owner
          Target      = target
          Arrived     = false
          Fighting    = false
          Alive       = true }
      add state.Fleets new_fleet }
```

The input script waits for the user to left click on a planet (the source) and then right click on another (the target) to send armies from one to the other if the source owner is the human player; care must be taken to allow the user to change source multiple times before selecting a target:

```
  let rec input_script () = {
      let rec wait_selection click_function =  {
        wait_condition click_function
        let mouse = mouse_position()
        let clicked = [p | p <- state.Planets &&
                           distance(p.Position,mouse) < 10.0]
        if clicked.Length > 0 then
          return clicked.[0]
        else
          do! wait_selection click_function }

      let rec wait_source_selection() = {
            let! source = wait_selection mouse_left_click
            if source.Owner = Human then return source
            else do! wait_source_selection() }

      let wait_target_selection =
            wait_selection mouse_right_click

      let rec wait_selection source = {
        let! res = wait_source_selection() ||
                   wait_target_selection()
        match res with
        | Left source -> wait_selection source
        | Right target -> return source,target }
      let! source = wait_source_selection()
      let! source,target = wait_selection source
      do mk_fleet source target Human
      do! input_script() }
```

The enemy AI selects a random pair of planets, one that belongs to itself and the other that belongs to the human and sends a fleet of ships from the first planet to the second:

```
  let rec enemy_ai () = {
     do! wait 1.0
     let own_planets = [p | p <- state.Planets&&p.Owner = AI]
     let enemy_planets = [p | p <- state.Planets&&p.Owner<>AI]
     if enemy_planets.Length > 0 && own_planet.Length > 0 then
       let source = own_planets[random(0,own_planet.Length-1)]
       let dest = enemy_planets[random(0,own_planet.Length-1)]
       if source.Armies > 0 then do! mk_fleet source dest AI
     do! enemy_ai() }
```

The main script of the game runs both the input and the AI in parallel:

```
  enemy_ai() && input_script()
```

4.5 CONSISTENCY AND AUTOMATED OPTIMIZATIONS

The asteroids game in particular shows two very important features of Casanova: *consistency* and *automated optimizations*.

Consistency comes into play when we are computing collisions between projectiles and asteroids. If we removed asteroids from the game state whenever we found those asteroids to be in collision with projectiles, then those asteroids would not be found anymore when determining which projectiles to remove. Let us consider this example:

```
asteroids = [a1; a2; a3]
projectiles = [p1; p2]
```

We check the asteroids for collision, and we find that `a2` collides with `p1`; we update the state so that:

```
asteroids = [a1; a3]
projectiles = [p1; p2]
```

Unfortunately now we do not find the collision between `p2` and `a2` anymore because `a2` has been removed from the set of current asteroids. While this specific bug is not too hard to fix once identified, it belongs to a larger class of common bugs that happen whenever the state of the game is only partially updated, and is representing part of the world at time step `t` and part at time step `t + 1`. If the state of the game is large and the order in which the various game entities are updated is not fixed (for example if sometimes asteroids are updated before projectiles, and other times the opposite), then this can become a major source of unpredictable errors that are difficult to reproduce, diagnose and fix. Casanova prevents these problems by applying a double buffering strategy that keeps the current state intact until all updates are performed. This way all computations act on a game state that represents the world at the same time step. Both the Casanova language and the Casanova libraries feature this transactional system.

The second important feature of Casanova is that of automated optimization of quadratic queries. All rules of the form:

```
rule Field : Table<T> =
  fun (state, self, dt) -> [ x | x <- state.Xs, cond x self ]
```

are optimized by building an index (be it a spatial partitioning index such as a QuadTree or a hash-map) that makes determining the Xs that satisfy the predicate cond much faster. Building this index can give dramatic performance increases. For the asteroid game, using this optimization gives a huge performance boost in a stress test: the framerate without the optimization is 8 fps, while the framerate with the optimization jumps to 577 fps (which is actually more than we expected). In similar stress tests, the RPG game goes from 14 fps to 217 fps, while the RTS goes from 27 fps to 202 fps.

Automated optimization is only available in the Casanova language at the moment. Since no rule writes the same memory location of another, rules can be computed in parallel, producing another noticeable speedup in the order of up to the number of available cores for certain programs.

5. 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, but it is now clear that we can try to tackle further tasks: *(i)* we are studying how to integrate a rendering engine into the Casanova system, so that by adding a few annotations to the game state (for example which model, texture or font is associated with each entity) then an appropriate rendering system (together with multi-threaded disk loading, etc.) can be instantiated by the library or the compiler; *(ii)* we are studying how to automate the definition of networking code for a host-client architecture by specifying which parts of the game state are updated on each client and how often they

are to be transmitted when their value changes and if their transmission is reliable or not; *(iii)* we are evaluating the possibility of integrating algorithmic skeletons for building standardized AIs (for example by using example-based methods such as [18]) and thus making the creation of engaging and challenging game entities simpler; *(iv)* we have a long list of query optimizations (taken from the databases literature [19]) that await implementation and integration into the game; *(v)* we plan on adding a reference counting mechanism to offload the garbage collector and recycle entities whenever possible, especially to aid hardware architectures such as the XBox 360 Power PC; *(vi)* we plan on explicitly implement a translator for coroutines from the monadic framework in which they are currently implemented into an explicit state machine to avoid the overhead associated with F# monads; *(vii)* we plan on compiling lists of structures into structures of lists to fully take advantage of processors such as those available in current generation consoles, which deal much better with arrays of simple values where the same computation is performed in order for each element [20].

6. 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. 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 and other fascinating tasks such as procedural generation rather than on the "nuts-and-bolts" of putting a game together and optimizing it. While Casanova is still in its early stages, our students and us have used it extensively and with good results, and we are certain that with further work the benefits of this approach will become much more apparent.

7. REFERENCES

[1] Entertainment Software Association. Industry Facts. (2010).

[2] Fullerton, Tracy, Swain, Christopher, and Hoffman, Steven. Game design workshop: a playcentric approach to creating innovative games. Morgan Kaufman, 2008.

[3] Ritterfeld, Ute, Cody, Michael, and Vorderer, Peter. Serious Games: Mechanisms And Effects. (2009), Routledge.

[4] Buckland, Mat. Programming Game AI by Example. (Sudbury, MA 2004), Jones & Bartlett Publishers.

[5] Wilson, Kyle. Inheritance vs aggregation in game objects. In http://gamearchitect.net/Articles/GameObjects1.html. (2002).

[6] Ampatzoglou, Apostolos and Chatzigeorgiou, Alexander. Evaluation of object-oriented design patterns in game development. In Journal of Information and Software Technology (MA, USA 2007), Butterworth-Heinemann Newton.

[7] Conal, Elliott and Hudak, Paul. Functional reactive animation. In International Conference on Functional Programming (ICFP) (1997), 263–273.

[8] Folmer, Eelke. 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 2007), Springer-Verlag, 66– 73.

[9] Giuseppe Maggiore, Alvise Spanò, Renzo Orsini, Giulia Costantini, Michele Bugliesi and Mohamed Abbadi. 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), Springer.

[10] Figueiredo, L. H. de, Celes, W., and Ierusalimschy, R. Programming advance control mechanisms with Lua coroutines. In Game Programming Gems 6 (2006), Mike Dickheiser (ed), Charles River Media, 357–369.

[11] Knuth, Donald E. The art of computer programming. (Redwood City, CA, USA 1997), Addison Wesley Longman Publishing Co., Inc.

[12] Walker White, Alan Demers, Christoph Koch, Johannes Gehrke, and Rajmohan Rajagopalan. Scaling games to epic proportions. In Proceedings of the 2007 ACM SIGMOD international conference on Management of data (SIGMOD) (New York, NY, USA 2007), ACM, 31–42.

[13] Costantini, Giulia and Maggiore, Giuseppe. Friendly F# (fun with game programming). (Venice, Italy 2011), Smashwords.

[14] Corp., Microsoft. The xna framework. (2004), http://msdn.microsoft.com/xna.

[15] Maggiore, Giuseppe. Casanova project page. In http://casanova.codeplex.com/ (2011).

[16] Maggiore, Giuseppe. Galaxy Wars Project Page. In http://vsteam2010.codeplex.com (2010).

[17] Richard Zhao, Duane Szafron. Generating Believable Virtual Characters Using Behaviour Capture and Hidden Markov Models. In 13th Internation Conference Advances in Computer Games (ACG) (Tilburg, Netherlands 2011), Springer.

[18] Garcia-molina, Hector, Ullman, Jeffrey D., and Widom, Jennifer. Database System Implementation. (1999), Prentice-Hall.

[19] Albrecht, Tony. Pitfalls of Object Oriented Programming. In International Conference Game Connect Asia Pacific (GCAP) (2009).

[20] Schuytema, Paul and Manyen, Mark. Game Development With LUA. In Game Development Series (Rockland, MA, USA 2005), Charles River Media, Inc.

[21] Leischner, Nikolaj, Liebe, Olaf, and Denninge, Oliver. Optimizing performance of XNA on Xbox 360. In FZI Research Center for Information Technology (2008), http://zfs.fzi.de.

[22] Wadler, Philip. Comprehending monads. In Mathematical Structures in Computer Science (1992), 61–78.