A formal specification for Casanova, a Language for Computer Games

Giuseppe Maggiore, Alvise Spanò, Renzo Orsini, Michele Bugliesi, Mohamed Abbadi, Enrico Steffinlongo

Università Ca' Foscari Venezia

DAIS - Computer Science

{maggiore,spano,orsini,bugliesi,mabbadi,esteffin}@dais.unive.it

ABSTRACT

In this paper we present the specification and preliminary assessment of Casanova, a newly designed computer language which integrates knowledge about many areas of game development with the aim of simplifying the process of engineering a game. Casanova is designed as a fully-fledged language, as an extension language to F#, but also as a pervasive design pattern for game development.

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, 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 AAA games) has created substantial interest in research on principled design techniques and on cost-effective development technologies for game architectures. We believe our proposal makes a significant step in this direction. Moreover, several teaching institutions are nowadays beginning the introduction of game development as a tool for engaging students in studying programming and better understanding computer science [4].

In this paper we will present the Casanova language and framework, which are built to simplify the creation of games while at the same time retaining the ability to build applications which are "more than toys". Casanova comes in two flavors: on one hand there is the language itself, but on the other there is a general methodology (aided by various libraries) for making games in existing languages. In this paper we describe the current state of the Casanova project, which has moved from the design phase of [5] into specification and implementation. In particular, we describe how the mechanisms behind Casanova work, from the syntax, type and semantics of our work in Section 2, to a discussion on implementation in Section 3 and to a final comparison of Casanova with other widely adopted frameworks such as C# and XNA to assess the effectiveness of our framework in Section 4.

1.1 RELATED WORK

The two most common game engine architectures found today in commercial games are object-oriented type hierarchies and component-based systems [6], [7]. These two more traditional approaches both suffer from a noticeable shortcoming: they focus exclusively on representing single entities and their update and draw operations in isolation, rather than by cooperating with all other entities. Also, behaviors that take longer than a single tick 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 the correct sequence of updates of the various entities of the game. To mitigate the difficulties of programming with this model, pre-existing game engines have been built to allow programming a game with a mixture of a visual programming language (called *editor*), plus a scripting language for further behaviors.

There are a few additional coding approaches for games that have emerged in the last few years as possible alternatives to traditional architectures: (functional) reactive programming [8], and SQL-style declarative programming [9]. FRP offers a solution to the problem of representing long-running behaviors, but 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 (SGL) use a lightweight, statically compiled and 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 all these issues in mind: the integration of the interactions between entities and long-running behaviors is seamless, and the resulting game world is always consistent. Furthermore, all the aspects of a game architecture can be integrated in a Casanova program: not just the game logic, but also rendering, networking, input management, etc.

2. ANATOMY OF A CASANOVA GAME

A Casanova program starts with the definition of the game world, that is a series of type declarations. Casanova does not require the developer to specify an update or a draw function; rather, the developer specifies a series of rules inside the various type declarations of the game entities. Rules describe how an entity (and its contents) changes value during a tick of the game loop. The update function will then consist of traversing the game state and building the new state by evaluating all the available rules. The draw function, similarly to the update function, traverses the state to fill a series of deferred batches. Whenever it encounters instances of any of the preset drawable entities (`DrawableModel`, `DrawableText`, `DrawableSprite`), the system adds each entity to its batch; each batch is then drawn with all its entities after traversal is complete.

After defining the game state, the developer defines its initial value. This initial value represents the starting state when the game is launched.

Rules are high-level, expressive constructs and being declarative they allow for many optimizations. 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 we have built an additional scripting system to specify imperative *processes* (through coroutines), which smoothly integrate with rules. Coroutines invoke each other with the `do!` and `let!` monadic operators [10]; 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 already, 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 [11].

The developer then defines the game scripts: the main script, plus a list of pairs of input scripts, where each pair is composed of an event detection script and an event response script. Whenever the first script detects an input event then the response is run.

Casanova supports mutable values through the type constructor `var`, and reference values which are not updated (since they are just references to values stored elsewhere in the state) through the type constructor `ref`.

2.1 Syntax

In the following is shown the syntax of a Casanova program: we start with the type definitions of the game state and the various game entities, each specifying the rules that define an update of the game state. Then we give the initial state, and finally we give the main and input scripts. Keep in mind that the initial state definition and the `GameState` type declaration do not show up explicitly in the grammar, as they are simply a let binding and a datatype declaration respectively and they are statically checked for existence after parsing:

```
Program ::= (Type-decl | Let-binding)* Expr
Type-decl ::= type Id [('a, ..)] = Type-body
Type-body ::= Type
           |  { Id [: Type] = Expr; .. } [with (Rule)+]
           |  Uid [of Type] | ..
Rule ::= rule Id = Expr
Type ::= 'abc.. | Id [(Type, ..)]
      |  Type * .. * Type | Type + Type
      |  ref Type | var Type
      |  script Type | table Type
Let-binding ::= let Pattern = Expr
             |  let rec Id = Expr and ..
Expr ::= Lit | Id | Uid | fun Pattern → Expr
      |  Let-binding in Expr | Expr.Id | Expr; Expr
      |  if Expr then Expr [else Expr]
      |  match Expr with Pattern → Expr | ..
      |  { MExpr }
Pattern ::= _ | Id | Uid [Pattern] | (Pattern, ..)
         |  Pattern as Id | (Pattern | Pattern)
         |  Pattern : Type
Lit ::= 123.. | 12.34.. | "string.." | 'c' | ()
     |  [Expr; ..] | [CExpr; ..] | { Id = Expr; .. }
     |  { Expr with Id = Expr; .. }
CExpr ::= for Pattern in Expr do CExpr
       |  Pattern in Expr | Expr | yield Expr
       |  if Expr then Expr else Expr
MExpr ::= repeat MExpr | wait Expr | run Expr
       |  return Expr | MExpr ==> MExpr
       |  MExpr && MExpr | MExpr || Mexpr
       |  Mexpr; Mexpr | Expr := Expr | yield
       |  let! Pattern = Expr in MExpr
       |  do! Expr | Let-binding in Mexpr
       |  match Expr with Pattern → MExpr | ..
       |  if Expr then MExpr [else MExpr]
Id ::= <any-case identifier>
Uid ::= <upper-case identifier>
```

For the sake of completeness, we included productions for all meaningful language constructs such as `let`, `if`, `fun` and in general all the terms usually found in a standard implementation of the ML language [12], from which Casanova derives strongly.

2.2 Casanova Type System

Note: in type rules, we denote type application according to the Casanova type syntax – i.e. the Haskell-style type application syntax where `T a` denotes the application of type parameter `a` to the parameterized type `T`. In F# excerpts we will instead use the .Net notation `T<'a>`.

The Casanova type system is very similar to one of the many known type systems of similar functional languages. We will not specify the typing rules for `if`, `let`, etc., as they are well known [12]. Casanova has two specific aspects that differentiate it from its cousin languages: how mutable variables, rules and scripts work. Rules are used to describe how a field, item or constructor of type `T`, defined inside a type definition for a type named `Entity`, is updated during a tick. Rules are thus associated with a function-term that defines how the next value for the rule will be computed during a tick of the update loop; this term takes as input the current game state, the value of the entity to which the rule belongs, and the delta time between the current and previous ticks. The term has thus type:

```
(GameState * Entity * float) -> T
```

`Entity` is the name of the parent type that contains the rule itself. For example, a valid rule may increment the position of an asteroid with time and with respect to its velocity:

```
type Asteroid = {
  Position : var Vector2; Velocity : Vector2 }
with rule Position = fun (state:GameState,self:Asteroid,dt) ->
  self.Position + self.Velocity * dt
```

Record fields bound to a rule appear simply as fields of the declared type, either `var` or not. They can therefore be read or even assigned accordingly. The rule function is used internally by the generated code in the `update` loop.

Values of type `var` can of course be accessed through the dereference unary operator `(!):var T->T` which is typed as:

$$\frac{\Gamma \vdash x : var\ T}{\Gamma \vdash !x : \mathrm{T}}$$

Assignment to vars are instead allowed only within scripts - Casanova in general controls effects by typing effectful computations as scripts. The typing rule for assignment is:

$$\frac{\Gamma \vdash x : var\ T,\ v : \mathrm{T}}{\Gamma \vdash x := v : Script\ Unit}$$

As shown by the language syntax, terms of kind *MExpr* offer all effectful constructs scripts have access to. Among those that introduce effects, `yield` suspends the current script for the remainder of the current tick and resumes it at the next tick, while `wait` suspends the current script for a certain amount of time:

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash yield : Script\ Unit} \qquad \frac{\Gamma \vdash t : float}{\Gamma \vdash wait\ t : Script\ Unit}$$

Scripts are sequenced together by binding them with either `let!` or `do!`, as in [F# monads]:

$$\frac{\Gamma \vdash t_1 : Script\ T \qquad \Gamma, x{:}T \vdash t_2 : Script\ T'}{\Gamma \vdash let!\, x = t_1\ in\ t_2 : Script\ T'}$$

When `t1 : Script Unit` then we can use `do!` instead of `let!`. To return a value from a script we use the return operator:

$$\frac{(\Gamma \vdash x : T)}{\Gamma \vdash return\ x : Script\ Unit}$$

The remaining combinators for scripts are `(&&)` to run two scripts in parallel, `(||)` to run two scripts concurrently (stop with the first to terminate), `(==>)` runs a script when another script finished with result `Some x` for some `x`, and finally `repeat` keeps running a script forever. As a simple example let us consider a script which waits for a flag to be turned on, and then it moves an entity right for 10 seconds:

```
{ return a.start_moving } ==>
  (repeat (a.p.x := a.p.x + 0.1) ||
   wait_condition { return a.p.x > 10.0 })
```

2.3 Semantics

The Casanova semantics is defined with two main goals in mind: consistency and performance. Consistency is needed to make sure that during each iteration of the update function the game state and all its contents represent values that belong to the same iteration; a large number of bugs in games come from manipulating a game state that is not fully updated. For example, consider an asteroids game where we wish to remove those pairs of asteroids and projectiles which are currently colliding. In this example, in-place update of the state can give undesired results when computing collisions between asteroids and projectiles:

```
[a1; a2; a3] [p1; p2] // a2, p1 collide
[a1; a3] [p1; p2]     // update asteroids
[a1; a3] [p1; p2]     // update projectiles
```

The result above should be `[a1; a3]` and `[p2]`.The bug above, while simplistic, is a more general instance of all those inconsistencies that arise from updates where some of the temporal invariants of the state are broken, that is the game state contains data that is part in the present and part in the future.

Good performance is needed to ensure that the update function executes as fast as possible, in order to make the game run at an interactive frame-rate. Performance is guaranteed by avoiding a "wasteful" semantics that would perform unnecessary computations, and by including important optimizations. We have defined Casanova in terms of how its programs are translated into equivalent F# programs. First, types are translated into (possibly imperative) F# types; then Casanova rules are used to build the update function and finally scripts are compiled into F# monads and run stepwise within the update loop. The generated F# program does not create a new game state at each tick of the update function, since this would allocate and discard too much memory (thus adding excessive overhead to the game runtime in terms of garbage collection). Still, purity makes it much easier to reason about our games, so we use a double buffering strategy for values resulting from the evaluation of rules: one slot is reserved for the value currently held by the rule (the value computed during the last tick), and the other to hold the next value, the one that is being computed during the current tick. The next value for each rule is only writable during each tick, while the current value is only readable; in effect, this gives the same result that we would have by generating a new game state, but without the overhead. Further optimizations are described in 2.3.4.

2.3.1 Type Translation

We define a transformation from Casanova types into F# types. The transformation mostly preserves the original structure: tuples remain tuples, records remain records, and so on. The only difference is that rules are represented with the special data-type `Rule<'a>`:

```
type Rule<'a> = { mutable Current : 'a; mutable Next : 'a }
```

When an entity is defined in terms of a rule, the `Rule` data-type is inserted into the entity together with a property that simplifies access to this field. For example when we write the following Casanova data-type:

```
type Ship = { Position : Vector2 = … } with rule Position …
```

it is turned into the F#:

```
type Ship = { _Position : Rule<Vector2> = … }
  member this.Position
    with get() = this._Position.Current
    and set p' = this._Position.Next <- p'
```

The body of the rule is ignored while generating types and it is used only to create the update function of the game.

2.3.2 Rules and Update

The `update` function is generated entirely by Casanova, and it evaluates all the rule functions associated with the game state definition and stores their result in the `Next` field of their rule. It is modeled as a polytypic function [13] (emulated through reflection and on-the-fly compilation) on the original game state; in the following we adopt the convention that T_{cnv} is the original Casanova type and $T_{F\#}$ is its transformation into F#. The `update` function simply traverses the state, and when it encounters a rule then it assigns to its `Next` field the result of evaluating the rule function. `update` is generated by a traversing the type definitions, starting from the game state and then one entity at a time. The function takes as input the type of the game state and returns a function that performs the update on its transformed F# type. In the following, we denote a type parameter as followed by the big arrow `=>`, and a regular parameter with the regular arrow `->`; a type parameter in this context can be analyzed with a switch-case:

```
update : $Type_{cnv}$ => $Type_{F\#}$ -> float -> Unit
```

`update` uses an auxiliary generator function which takes as input the type of the game state, the type of the current entity and the type of the field in the entity that is being updated; this auxiliary function is called `update'` and has type:

```
update' : $GameState_{cnv}$ => $Entity_{cnv}$ => $T_{cnv}$ =>
          $GameState_{F\#}$ -> $Entity_{F\#}$ -> $T_{F\#}$ -> float -> Unit
```

The update function simply invokes the `update'` function; since at the start of the generation of the update function the state is the entity we are processing, we invoke update' by passing the state three times: one as the state, one as the current entity and one as the current field. Type parameters are written between square brackets `[∘]`:

```
update [$GameState_{cnv}$] (s:$GameState_{F\#}$) (dt:float) =
  update' [$GameState_{cnv}$] [$GameState_{cnv}$] [$GameState_{cnv}$] s s s dt
```

When we encounter a primitive or a reference value then we do nothing:

```
update' [S] [E] [P] s e v dt = ()
update' [S] [E] [ref T] s e v dt = ()
```

When we are processing a variable inside an entity `E` then we proceed by updating the contents of the variable. If the type parameter `T` of the variable is a type declaration, that is it has a name, then the processed value becomes the current entity:

```
update' [S] [E] [Var T] s e v dt =
  if T is not a type decl then update' [S] [E] [T] s e v dt
  else update' [S] [T] [T] s v dt
```

When we encounter a tuple (or, similarly, a record) then we update all its internal values:

```
update' [S] [E] [$T_1$ * ... * $T_n$] s e ($v_1$,...,$v_n$) dt =
  if $T_1$ is not a type decl
    update' [S] [E] [$T_1$] s e $v_1$ dt
  else
    update' [S] [$T_1$] [$T_1$] s $v_1$ $v_1$ dt
  …
```

When we update a discriminated union then we pattern match on the updated value and update the parameter of the current constructor:

```
update' [S] [E] [$T_1$+$T_2$] s e v dt =
 match v with
 | Left $v_1$ ->
   if $T_1$ is not a type decl then
     update' [S] [E] [$T_1$] s e $v_1$ dt
   else
     update' [S] [$T_1$] [$T_1$] s $v_1$ $v_1$ dt
  | Right $v_2$ ->
   if $T_2$ is not a type decl then
     update' [S] [E] [$T_2$] s e $v_2$ dt
   else
     update' [S] [$T_2$] [$T_2$] s $v_2$ $v_2$ dt
```

Similarly, a list is updated by iterating and updating all its elements. Finally, if the update function encounters a rule, then the rule body is evaluated and its value is updated, stored in the state (since rules may be assigned in the transformed F# data-types) into `Next` and then the `Current` value is updated; we update `Current` rather than `Next` value for consistency (since `Next` is treated as a write-only value during a tick):

```
update' [S] [E] [rule T = term] s e v dt =
  v.Next <- term s e dt
  if T is not a type decl
    update' [S] [E] [T] s e v.Current dt
  else
    update' [S] [T] [T] s v.Current v.Current dt
```

The update function does not traverse functional terms or scripts. When the update function has finished performing its work, then it traverses the game state and swaps all the `Current` and `Next` fields of each rule, since the `Next` field is now fully computed and contains the latest value of the rule. The definition of the function that performs the swap of `Current` and `Next` for each rule is omitted as it is very similar to the definition of the update function seen above: this function iterates all entities recursively starting from the game state and whenever it encounters a rule it swaps its `Current` and `Next` fields.

2.3.3 Scripts

Scripts are compiled into F# monads. For a more comprehensive treatment of this mechanism, see [14]. The various scripting constructs are translated with a one-by-one correspondence into our monad. Since scripts represent computations that may be suspended and resumed, we implement such a coroutine system.

The monadic data-type that we use represents a script as a function that performs a step in the computation of the script. This function, when evaluated, performs some side-effects on the state, and then it either returns the final result if the script has finished computing, or else it returns the continuation of the script:

```
type Script<'a> = Unit -> Step<'a>
and Step<'a> = Done of 'a | Next of Script<'a>
```

Returning simply encapsulates a value around the `Done` data constructor:

```
let return(x:'a) : Script<'a> = fun () -> Done x
```

Binding runs a script until it returns a result with `Done`. When this happens, the result of the first coroutine is passed to the second coroutine, which is then run until it completes:

```
let bind (p:Script<'a>, k:'a->Script<'b>)
  : Script<'b> =
  fun () ->
    match p () with
    | Done x -> k x ()
    | Next p' -> Next(this.Bind(p',k))
```

Yield suspends and then returns nothing:

```
let yield : Script<Unit,'s> = fun s -> Next(fun s -> Done ())
```

The above functions (`bind`, `return`, and `yield`) are a complete definition of a fully functional monad; they cover the `let!`, `do!`, `return` and `yield` constructs of the Casanova language.

At each tick of the simulation each active script is run for a single step by passing it a `()` value and by replacing it (for the next iteration) with its own result, that is its continuation; if the script finishes then it is removed from the list of active scripts.

2.3.4 Optimization
A great deal of development effort in modern games is spent working on editing the game source, but rather than adding new and useful features the same code is tuned until it is efficient enough, by applying various optimizations such as visibility culling (to reduce the number of rendered models) and other techniques. One of the original design goals of Casanova is to save developers time and effort by automatically performing several of those optimizations that would otherwise be hand-written.

A lot of the effort in game optimization goes into optimizing quadratic queries [15]; many games feature lots of searches to compare two collections: collision detection, visibility, interaction, etc. Let's consider example of such a query when finding the asteroids that collide with a projectile in an asteroid shooter game; this query would be computed for each projectile, and thus its overall complexity in a naïve implementation would be $O(n_{ast} \times n_{proj}) = O(n^2)$. By using a spatial partitioning index on the asteroids, it becomes possible to evaluate this query in a much shorter time. Quadratic queries in Casanova generate optimized code by adding an appropriate index to the game state and solving predicates faster by taking advantage of this index in all related queries [16]. Another important optimization is that of avoiding completely the rule swapping routine at the end of the update function. To avoid this, we modify the rule data-type as follows:

```
type Rule<'a> = { Values : 'a[]; Index : ref<int> }
  member this.Current with get() = Values.[!Index % 2]
  member this.Next with set v' = Values.[(!Index+1)%2] <- v'
```

With this implementation, swapping the various current and next values inside all the rules of the game simply requires incrementing the `Index` reference, which is a global value shared among all rules. Collections inside rules can be optimized as well with a simple modification. Instead of constantly creating new collections at each tick of the update function, collection rules are optimized by pre-allocating two mutable collections (the F# data-type is `ResizeArray`). When computing the new value of the collection rule the `Next` collection is cleared and the values of the new collection are added to it.

The final optimization that is performed by Casanova is parallel evaluation of rules in different thread, since no two rules can write the same memory location.

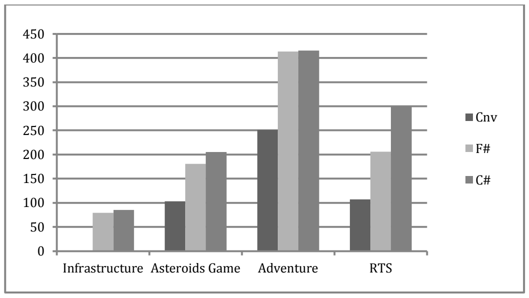
2.4 Case Study
As a test-bed for Casanova, we have built an actual game with a group of MSc students. The game, Galaxy Wars, is a real-time strategy game where the player conquers a star sector by building and using fleets of starships. It may be found at [17], both the sources and the latest executable. The game features more than a hundred thousand lines of code, and supports hundreds of interacting entities and up to eight players across LAN and Internet. This project has allowed us to see rules, scripts and most of Casanova in action in a much larger work than the usual smaller samples of code we have worked with [18], and from the feedback of this project many aspects of Casanova have taken their final shape. The game is also in the process of being published commercially.

3. IMPLEMENTATION
It is important to stress out that while we have designed Casanova as a fully-fledged programming language (the compiler of which is currently under construction), considering it just from this point of view is reductive. Casanova is more importantly a design methodology for making games, which covers the definition of the game state, and of the update and draw functions. At the time of writing there are two implementations of Casanova that cover the aspects of the language presented above in various manners. The first implementation is simply as an F# library. Said library contains an implementation of the scripting monad and an implementation of rules. Drawing and updating the state are done automatically through highly optimized reflection. While this library does not support everything about Casanova (for example rules require the use of specialized data-types), the fact of it being a library makes it flexible and easy to use in many contexts. We show how to use the library to implement various games in [18]. Moreover, the library takes full advantage of all the development tools built to support F#. This library is currently being used and extended organically in the Galaxy Wars research game [17] and in a series of smaller sample games [18]. The second implementation is a C++ meta-programming library which implements state traversal, updating and drawing, and coroutines with a rather articulated system of partially specialized templates. The renderer for the F# library is built in XNA, while the renderer for the C++ library is built with DirectX 10.

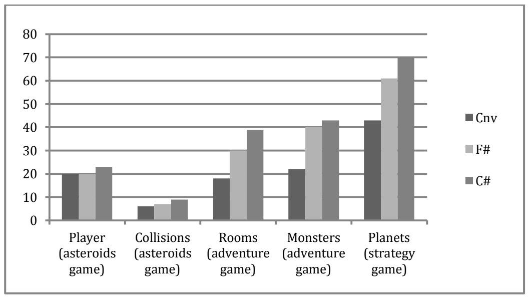
Both systems are still a work in progress, but they may be tested, experimented with and extended into further projects already.

4. 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. Similarly, there exist no metrics that make it clear that a language is good for a certain job, or even to compare that a language is better than another at that job. It is with this in mind that we proceed with exposing a series of arguments in answer to our original claim that Casanova is better suited than traditional mainstream object-oriented languages such as C# (plus libraries such as XNA) for real-time game development. We have chosen C# as it is widely used in the game industry (Xbox, iOS, Windows Phone 7, Android) and because it is widely used as a "simpler" game development language when compared with the industry standard of C++. We will discuss how Casanova programs are overall much shorter than equivalent C# programs (measured excluding lines containing only parentheses such as `{` and `}` or trivial code such as property declarations and such), and we will also discuss how a series of "typical" snippets of game code taken from three different games (an asteroid shooter game, an action/adventure game and a strategy game [18]) compare between Casanova, the Casanova F# library, and idiomatic C#.

The first comparisons that we make 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:



We now move to 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 just listed, 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 removes the need for boilerplate code, it removes the need to traverse the game world to update and draw each entity, and it allows to express the logic of the game with fitting abstractions.

We have also performed a series of benchmarks on a single game implemented in Casanova to test the gains obtained by the various types of optimizations available in term of ticks per second that the game becomes able to perform. We compare the optimized Casanova programs with their un-optimized version to assess the effectiveness of each optimization, but notice that the un-optimized Casanova version is compiled into F# code that offers roughly (within 5% in all our tests) the same performance of equivalent C# code:

Optimization	FPS	% gain
None	0.375	N/A
Fast rule swap	0.387	103%
No realloc for lists	0.387	103%
Rule threads	0.782	203%
Query	213	> 10000%
All together	233	> 10000%

As we can see, the various optimizations each offer some speedup, but parallelism and query optimization do the most. It is also important to keep in mind that these optimizations require no work on the part of the Casanova developer. The sources used for the benchmark are those of the RTS game in [18].

5. CONCLUSIONS

Games and multimedia applications are extremely widespread. Disciplined models and techniques that simplify game development have a unique chance of having a significant impact by allowing the creation of games without needing to focus on lots of complicated details that are not really related to the game itself. Casanova is a step in this direction, and the framework is shaping up as to cover the creation of a complex game logic, plus flexible input management, and drawing.

6. 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. White, Li Ty and Alice Team R and Y Pausch (head and Tommy Burnette and A. C. Capehart and Dennis Cosgrove and Rob Deline and Jim Durbin and Rich Gossweiler and Koga Jeff. A Brief Architectural Overview of Alice, a Rapid Prototyping System for Virtual Reality.
5. 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.
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. 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.
8. Conal, Elliott and Hudak, Paul. Functional reactive animation. In International Conference on Functional Programming (ICFP) (1997), 263–273.
9. 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.
10. Costantini, Giuseppe Maggiore and Giulia. Friendly F# (fun with game programming). (Venice, Italy 2011), Smashwords.
11. 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.
12. Pierce, Benjamin. Types and Programming Languages. MIT Press, Cambridge, Massachusetts , 2002.
13. Jeuring, Patrik Jansson and Johan. PolyP - a polytypic programming language extension. (1997), Symposium on Principles of Programming Languages (POPL).
14. Giuseppe Maggiore, Michele Bugliesi and Renzo Orsini. Monadic Scripting in F# for Computer Games. (Oslo, Norway 2011), Harnessing Theories for Tool Support in Software (TTSS).
15. Buckland, Mat. Programming Game AI by Example. (Sudbury, MA 2004), Jones & Bartlett Publishers.
16. Garcia-molina, Hector, Ullman, Jeffrey D., and Widom, Jennifer. Database System Implementation. (1999), Prentice-Hall.
17. Maggiore, Giuseppe. Galaxy Wars Project Page. In http://vsteam2010.codeplex.com, http://galaxywars.vsteam.org.
18. Maggiore, Giuseppe. Casanova project page. In http://casanova.codeplex.com/.