Designing Casanova: a language for games

G. Maggiore, A. Spanò, R. Orsini, G. Costantini, M. Bugliesi, M. Abbadi

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

Abstract. Games are extremely complex pieces of software which give life to animated virtual worlds. Game developers carefully search the difficult balance between quality and efficiency in their games.
In this paper we present the Casanova language. This language allows the building of games with three important advantages when compared to traditional approaches: simplicity, safety and performance. We will show how to rewrite an official sample of the XNA framework, resulting in a smaller source and higher performance.

1 Introduction

Computer games promise to be the next frontier in entertainment, with game sales being comparable to movie and music sales in 2010 [2].

This unprecedented market prospects and potential for computer-game diffusion among end-users has created substantial interest in research on principled design techniques and on cost-effective development technologies for game architectures. Our present endeavour 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 [8].

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, components, reactive progamming, etc have all been used with some degree of success for this purpose [10, 3, 9].

In this paper we will present the Casanova language, a language for making games. Casanova offers a mixed declarative/procedural style of programming

which has been designed in order to facilitate game development. The basic idea of the language is to require from the developer only and exclusively those aspects of the game code which are specific to the game being developed. The language aims for simplicity and expressive power, and thanks to automated optimizations it is capable of generating code that is much faster than hand-written code and at no effort for the developer. The language offers primitives to cover the development of the game logic, and incorporates the typical processing of a game engine. Also, the language is built around a theoretical model of games with a "well-formedness" definition, in order to ensure that game code is always a good model of the simulated virtual world.

In the remainder of the paper we show the Casanova language in action. We begin with a description of the current state of game engines and game programming in Section 2. In Section 3 we define our model of games. We describe the Casanova language in Section 4. We show an example of Casanova in action, and also how we have rewritten the game logic of an official XNA sample from Microsoft [6] in Casanova with far less code and higher runtime performance in Section 5. In Section 6 we discuss our results and some future work.

2 Background

In this section we discuss some current approaches to game development.

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 [7].

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 [10].

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.

There are two more 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 [9]) is often studied in the context of functional languages. FRP programming is a data-flow approach where value modification is automatically propagated along a dependency graph that

represents the computation. FRP offers a solution to the problem of representing long-running behaviors, even though it does not address the problem of many entities that interact with each other.

SQL-queries for games have been used with a certain success in the SGL language (see [18]). This approach uses a lightweight, statically compiled query engine for defining a game. This query engine is aggressively 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 two issues in mind: with Casanova, the integration of the interactions between entities and long-running behaviors is seamless.

3 A Model for Games

We define a game as a triplet of a game state, an update function and a series of asynchronous behaviors. In this model we purposefully ignore the drawing function, since it is not part of the current design of Casanova.

```
type Game 's =
  { State : 's; Update : 's -> DeltaTime -> (); Behavior : 's -> () }
```

The game state is a set of (homogeneous) collections of entities; each entity is a collection of attributes, which can either be *(i)* primitive values, *(ii)* collections of attributes or *(iii)* references to other entities.

The update function modifies the attributes of the entire state according to a fixed scheme which does not vary with time; we call this fixed scheme the **rules** of the game; rules can be physics, timers, scores, etc. Each attribute of each entity is associated to exactly one rule. The update function is quick and must terminate after a brief computation, since it is invoked in a very tight loop that should perform 60 iterations per second.

The behavior function is a sequential process which performs a series of operations on the attributes of the game entities. It is a long-running, asynchronous process with its own local state, it runs in parallel with the main loop and it can access the current clock time at any step to perform actions which are synchronized with real time. The processing over the game state that a behavior performs usually takes more than one tick; behaviors are used, for example, for implementing AIs.

A game engine is thus a certain way of processing a game:

```
let run_game (game:Game 's) =
  let rec run_rules (t:Time) =
    let t' = get_time()
    game.Update game.State (t'-t)
    run_rules t'
  parallel (run_rules (get_time()), game.Behavior(game.State))
```

We define four properties of a correct and well-behaved game: *(i)* each entity is updated exactly once per tick, *(ii)* entity update is order-independent, *(iii)* tick always terminates and *(iv)* the game runs at an interactive framerate.

Casanova guarantees only the first three requirements. The fourth requirement cannot be guaranteed, since it heavily depends on factors such as the size of the virtual world and the computational resources of the machine used to run the game; nevertheless, by automating certain optimizations Casanova makes it easier to achieve the fourth requirement.

3.1 Architecture of a Casanova Game

We now discuss the architecture of a Casanova game.

Behaviors are used to make it easier to build complex input, articulated level logics or customized AI algorithms into the game. While Casanova does not (yet) integrate any deduction engine or proper AI system, it makes integrating such a system with the game loop and the game state much simpler.

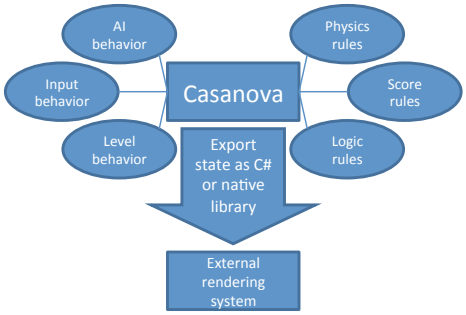

Fig. 1. Game architecture with Casanova

Rules are used to build all the regular logic that the game continuously repeats, for example the fact that when projectiles collide with an asteroid then the asteroid is damaged or other logical relationships between entities. Rules are the main workhorse of a game, and Casanova ensures that all the queries that make up the various rules maintain the integrity of the state and are automatically optimized to yield faster runtime.

The Casanova compiler will then export the game state as a series of type definitions and classes that can be accessed directly (that is without any overhead) from a C# or C++ rendering library; this way existing rendering code and engines can be integrated with Casanova with little effort.

4 The Casanova language

In this section we present the Casanova language; for a more detailed treatment, see [14]. Casanova is inspired to the ML family of languages.

4.1 Design Goals

We have designed the Casanova language with multiple goals in mind. First of all, Casanova games must be easy and intuitive to describe. For this reason we have used a mix of declarative and procedural programming. For expressing rules,

declarative programming is simple, allows the developer to focus on what he wants to achieve rather than how, and there is a wealth of powerful optimization techniques for declarative operations on sequences of values coming from the field of databases [11]. The declarative portions of a game are all executed in parallel, and can take advantage of multi-core CPUs.

Procedural programming, in particular coroutines [12], are used to describe computations that take place during many ticks of the game engine. Imperative coroutines are used to express the behaviors of a game. These behaviors are executed sequentially and with no optimizations, since they can access any portion of the state both for reading and writing, and they may perform any kind of operation.

4.2 A brief introduction to Casanova

Casanova is a programming language designed aroung a set of core principles aimed at aiding game development. Here we describe the language "at a glance", by listing its features designed to simplify repetitive, complex or error prone game coding activities: *(i)* Casanova integrates the game loop and time as first-class constructs. The game loop and time management are almost always an important part of game development libraries, for example see [5]; *(ii)* it performs a series of optimizations that are usually found hand-coded in virtually all game engines [8], such as logical optimization of queries on lists and spatial partitioning/use of indices to speed up quadratic queries such as collision detection (for example: `colliders(self) = [other | other <- Others, collides(self,other)]`; *(iii)* it guarantees that updates to the game state during one tick are consistent, that is the state is never partially updated thanks to a (high-performance) transactional system; *(iv)* it offers a scripting system that integrates seamlessly with the update loop.

We have designed Casanova with the aim of adding more features such as: *(i)* automated generation of all the rendering code; *(ii)* automated generation of all the networking code; *(iii)* automated generation of all or parts of an AI system.

Of course the language can also serve as a general purpose language. Any application that requires performing computations and visualization on a complex set of data which evolves over time according to a set of fixed rules might benefit from using Casanova. In the future we may investigate other possible uses of the language in this direction. As a final remark,it must be noted that Casanova sometimes constrains the developer; for example, at most one rule may be associated with any given field of the game state and rules are always applied at every tick of the simulation. Since developers may find this set of restrictions too tight we have included a scripting system which can also act as a "wildcard" in this regard, that is scripts have essentially no limitations in expressivity (scripts are a general purpose programming language with coroutines) and for this reason they can be used to express anything that the rule system cannot, albeit renouncing various useful features such as automated optimization.

4.3 Syntax, Semantics and Types

The details of the Casanova language syntax, semantics and type system are defined in [14]. In this subsection we give a general overview of the most salient aspects of the language.

A Casanova program is divided into three parts: *(i)* the state definition, *(ii)* the initial state and *(iii)* the main behavior.

The state definition contains the type definitions of the game state and game entities, together with the rules the various fields are subjected to. Rules may be nested, that is a field may contain a rule of type `Rule T`, where `T` contains a value of type `Rule V`. This is quite common, and we will seen an instance of this in the example below (in the `Introductory Example` subsection).

Entities and the state may be defined in terms of the usual type constructors found in a functional language: records, tuples and discriminated unions. Also, we can define values of type: table (for sequences), variable (for mutable cells), rule (for updateable fields) and reference (for read-only pointers).

The initial state defines the starting value of the various game entities.

The main behavior is an imperative process which runs for the entire duration of the game. A behavior may spawn (`run`) other behaviors, suspend itself for one or more ticks (`yield` or `wait`) or wait for another behavior to complete before resuming its execution (`do!` or `let!`). In addition behaviors may access the state without any limitation; a behavior can read or write any portion of the state: `:=` is the assignment operator and `!` is the lookup operator.

Behaviors can be combined with a small set of operators that define a simple concurrent calculus: `parallel x y`, which runs two behaviors in parallel and returns the pair with their results; `concurrent x y`, which runs two behaviors in parallel and returns the result of the first to terminate; `x => y`, which runs behavior `y v` only when `x` terminates with result `Some v`; and `repeat x`, which continuosly runs a behavior.

The tick function of the game is built automatically by the Casanova compiler, and it executes all running behaviors until they `yield`; then it executes all rules (in parallel and without modifying the current game state to avoid interferences); finally it creates the new state from the result of the rules.

Rules do not interfere with each other, since they may not execute imperative code. If rules immediately modified the current state, then their correctness would depend on a specific order of execution. Specifying said order would place an additional burden on the programmer's shoulders.

The tick function for rules presents a problem which is partly addressed with `references`: portions of the state must not be duplicated, for correctness reasons. This means that each entity in Casanova may be subjected to some rules but only once; if an entity is referenced more than once then it may be subjected to more (and possibly even contradictory) rules. For this reason we make any value of type `Rule` (or which contains a field of type `Rule`) linear [16]. This means that a value of type `Rule T` may be used at most once, and after it is read or used it goes out of scope.

We use the type constructor `Ref T` to denote a reference to a value of type `T`. A reference is a shallow copy to an entity which primary value is stored elsewhere. This allows for the explicit sharing of portions of the game state without duplication of rules, since rules are not applied to references. This also allows for safe cyclical references, such as:

```
type Asteroid = { ... Colliders : Rule(Table(Ref Projectile)) }
type Projectile = { ... Colliders : Rule(Table(Ref Asteroid)) }
```

This restriction is enforced statically during type checking, and it ensures that all rules are executed exactly once for each entity.

The type checker enforces another property: a behavior gives a compile-time error unless it is statically known that all code paths yield. This is achieved by requiring that `repeat` and `=>` are never invoked on a behavior which does not yield in all its paths. For example, behaviors such as:

```
repeat { if !x > 0 then yield else y := 10 }
```

generate a compile-time error.

This ensures that the tick function will always terminate, because rules are non-recursive functions and behaviors are required to never run without yielding indefinitely.

Of course it is possible to lift this restriction, since it may give some false negatives; for this reason, the actual Casanova compiler will be configurable to give just a warning instead of an error when it appears that a script does not yield correctly, to leave more freedom to those developers who need it.

So far the Casanova language enforces the following properties:

- developers do not have to write the boilerplate code of traversing the state and updating its portions; this happens thanks to the fact that Casanova automatically builds the game loop
- all entities of the state are updated exactly once (even though they may be shared freely across the state as `Ref`s); this happens thanks to the linearity of the `Rule` datatype and the automatic execution of all rules by the game loop
- rules do not interfere and are processed simultaneously; this happens thanks to the linearity of the `Rule` datatype and thanks to the fact that the state is created anew at each tick
- the tick function always terminates; this happens because the state is not recursive (again, thanks to the linearity of `Rule`) and because our coroutines are statically required to always invoke `yield`

These properties alone are the correctness properties and ensure that the game will behave correctly. We will now see an example Casanova game. We will also see the set of optimizations implemented by the Casanova compiler, that make sure that a game runs fast with no effort on the part of the developer.

4.4 Introductory Example

A Casanova program starts with the definition of the game state, the various entities and their rules. A field of an entity may have type `Rule T` for some type `T`. This means that such field will contain a value of type `T`, and will be associated with a function of type: $\texttt{Ref}(\texttt{GameState}) \times \texttt{Ref}(\texttt{Entity}) \times \texttt{T} \times \Delta\texttt{Time} \rightarrow \texttt{T}$

This function is the *rule function*, and its parameters are (they can be omitted by writing an underscore `_` in their position) *(i)* the current state of the game; *(ii)* the current value of the entity we are processing; *(iii)* the current value of the field we are processing; *(iv)* the number of seconds since the last tick.

When a field does not have an explicit rule function, then the identity rule is assumed. A rule function returns the new value of a field, and cannot write any portion of the state. Indeed, the current value of the state and the current entity are readonly inside the body of a rule function to avoid read-write dependencies between rules.

Updating the state means that all its rule functions are executed, and their results stored in separate locations. When all rule functions are executed, then the new state is assembled from their results.

In the remainder of the paper we will omit some type annotations; this is possible because we assume the presence of type inference.

In a field declaration, the `:` operator means "has type", while the `::` operator specifies the rule function associated with a rule.

The `!` operator is the dereferencing operator for rules, and it has type `Rule T -> T`.

Let us show how we would build a very simple game where asteroids fall down from the screen and are removed when they reach the bottom of the screen:

```
type Asteroid = {
    Y     : Rule float :: fun (_,self,y,dt) -> y + dt * self.VelY
    VelY  : float
    X     : float }

type GameState = {
    Asteroids
        : Rule(Table Asteroid)
        :: fun (_,_,asteroids,_) -> [a | a <- asteroids && a.Y > 0]
    DestroyedAsteroids
        : Rule int
        :: fun (_,self,destroyed_asteroids,_) -> destroyed_asteroids + count
            ([a | a <- !self.Asteroids && a.Y <= 0]) }
```

In the state definition above we can see that the state is comprised by a set of asteroids which are removed when they reach the bottom. Removing these asteroids increments a counter, which is essentially the "score" of our pseudo-game. Each asteroid moves according to its velocity.

The initial state is then provided:

```
let state0 = { Asteroids = []; DestroyedAsteroids = 0 }
```

Behaviors in Casanova are based on coroutines, that is they are imperative procedures which may invoke the `yield` operator. Yielding inside a behavior suspends it until the next tick of the game. Behaviors may freely access the

state for writing, that is behaviors are less constrained than rules but for this reason they also support less optimizations. The only requirement that Casanova enforces in behaviors is that they must never iterate indefinitely without yielding, and this requirement is verified with a dataflow analysis.

When the main behavior of a game terminates, the game quits.

The main behavior of our game spawns asteroids every 1-3 seconds until the number of destroyed asteroids reaches 100. The main behavior of our game is defined as:

```
let main state =
  let rec behavior() = {
      do! wait (random.Next(1,3))
      state.Asteroids.Add { X = random(-1,+1); Y = 1; VelY  = random
          (-0.1,-0.2) }
      if !state.DestroyedAsteroids < 100 then do! behavior() else return () }
  in behavior()
```

The imperative syntax loosely follows the monadic [15, 17] syntax of the F# language, where a monadic block is declared within `{}` parentheses, and combining behaviors is done with either `do!` or `let!` and returning a result is done with the `return` statement. This allows us to clearly mark the points where a behavior waits for another behavior to complete before taking its result and proceding.

4.5 Optimization

Casanova is designed to make it easy to automatically perform three main optimizations: memory recycling, rule parallelization and query optimization.

Memory recycling, is a simple yet effective optimization for all those platforms (such as the Xbox 360) with a slow garbage collector [4]. Memory recycling means that `Rule T` fields allocate a double buffer for storing both the current and the next value for rules, instead of regenerating a new state at each tick. Rule parallelization is made possible by the static constraint that rules are linear: this means that no rules write the same memory location. We also know that rules may not freely write any references. These two facts guarantee thread safety, that is we may run all rules in parallel. The final optimization is query optimization. Nested list comprehensions (also known as "joins" in the field of databases [11]) can have high computational costs, such as $O(n^2)$, for example when finding all the projectiles that collide with asteroids. Such a complexity is unacceptable when we start having a large number of asteroids and projectiles, because it may severely limit the maximum number of entities supported by the game. We use the same physical optimization techniques used in modern databases: we build a spatial partitioning index (such as a quad-, oc-, R-, etc. tree) to speed up our collision detection. The resulting complexity of the same query with a spatial partitioning index is $O(n \log n)$, which executes much faster and allows us to support larger numbers of entities.

4.6 A Full Example

We now show a full example of a game where a series of balls are thrown from one side of the screen and bounce towards the other side; the balls are removed when they reach the other side of the screen.

We start by defining the state (a collection of balls) and its rules (gravity, motion and removal of those balls that reach one side of the screen):

```
let g = Vector2(-9.81,0.0)

type BallsState = {
    Balls     : Rule(Table Ball))
              :: fun (_,_,balls,_) -> [b | b <- balls && b.X <= 50.0 ] }
type Ball = {
    Position  : Rule Vector2
              :: fun (_,ball,p,dt) ->
                     if p.Y < 0.0 then Vector2(p.X, 0.0)
                     else p + !ball.Velocity * dt

    Velocity  : Rule Vector2
              :: fun (_,ball,v,dt) ->
                     if p.Y < 0.0 then Vector2(v.X, -v.Y) * 0.6
                     else v + g * dt }
```

Then we define the initial state, which does not contain any balls:

```
let state0 = { Balls = [] }
```

Finally we define the main behavior which launches the balls, one every second:

```
let rec main state = {
    do! wait 1.0
    state.Balls.Add { Position = Vector2(0.0, 0.0); Velocity = Vector2(5.0,
         10.0) }
    do! main state }
```

5 Case Study

In this section we will describe how we have rewritten the XNA Spacewar [6] sample in Casanova, the resulting reduction in code and the increases in performance obtained. We have chosen Spacewar because it is small enough to be didactically useful while being built as a starter kit, that is a starting point to be edited and extended into a different game and not just as a sample or tutorial; from this point of view Spacewar should be considered as a small, yet complete and well-built, game.

The Casanova compiler is still in its very early stages, and as such it is not yet ready for the task. The definition of the compiler can be followed by hand, and since the first Casanova compiler will generate F# code, we have written such code by hand as the compiler would have output it.

5.1 Rewriting the Game

The original sample features two ships that shoot each other while dodging asteroids that float around the gaming area. A star in the center of the playing field pulls the players with its gravity. The first player to destroy the other (by hitting him or by making him crash on another celestial body) wins the stage.

The game state is defined as the two players, their ships, the table of asteroids and projectiles and the sun. Also, the state contains the current gameplay status, which can either be `Playing` or `GameOver w` where `w` is the winner.

The source code of the original sample plus our implementation can be found in [1]; the current implementation of the Casanova compiler is incomplete, and at the time of writing the type checker and the F# code generator are both producing their first correct outputs but are not yet integrated together. The details of the porting are discussed in detail in [14], and we omit them here for reasons of space.

We have slightly modified the original sample so that testing could be automated. For this reason we have removed the 30 seconds time limit of each level, we have removed the victory and ending conditions, we have automated ships movement and shooting and we have increased the maximum number of asteroids and projectiles to 12 and 200 respectively. This way we have obtained an automated stress test.

We have also removed all rendering features, to avoid benchmarking rendering algorithms: Casanova does not generate rendering code, so such a comparison would have been meaningless; also, Casanova can be integrated with the very same C# rendering code of the original Spacewar. We compare the resulting framerates to see how many simulation steps per second the original game logic is capable of performing versus the number of steps per second of the Casanova game logic; the higher this number, the more efficient the game logic and the more time remains for each frame to perform complex rendering.

As a final remark, it is worth noticing that while the original sample includes more than one thousand lines of code the length of the corresponding Casanova program is 348 lines long. The Casanova source easily fits a few pages, while navigating the original source may prove a bit complex because of its sheer size.

5.2 Resulting Benchmarks

We have benchmarked the modified sample on both the Xbox 360 and a 1.86 Ghz Intel Core 2 Duo with an nVidia GeForce 320M GPU and 4GB of RAM. In the table below we can see the framerates of the various tests.

C# XBox	C# PC	Casanova XBox	Casanova PC
8	9	22	577

Table 1. Framerate of the original Spacewar vs the Casanova implementation

As we can see, full Casanova optimization always beats the original source by at least a factor of 2. The Xbox implementation suffers from the generation of garbage, which is a known problem of the XNA implementation on the console ([4]); indeed, profiling the garbage collector shows that large amounts of temporary memory are being generated by the program. It is noticeable that on the PC, thanks to the full optimizations done by Casanova, performance increased by almost two orders of magnitude: such an impressive increase was quite unexpected even by us, more so when keeping in mind that those optimizations will be automated by the compiler.

6 Conclusions

In this paper we have presented the design of the Casanova language, a hybrid declarative/procedural language for making games. The language has a triple focus on simplicity and correctness (to increase developer productivity, given the complexity of game development) and performance (to ensure high framerates).

We have defined a model that generalizes an abstract game, and we have introduced four important properties that describe a good game. We have shown how the Casanova language respects these properties, that is:

- rules are applied exactly once for each entity
- rules are order-independent
- ticks always terminate
- automated optimizations ensure fast execution

Our first goal is to implement a fully working prototype of the Casanova compiler that outputs F# code. The compiler is still in its very early stages, and a lot of work is still needed to achieve this goal.

Further (and less obvious) improvements may be adding support for rendering, networking and (fully or partially) automated AI. Another venue that we are investigating is the support for reusable libraries of ready-made components, possibly with some form of statically resolved polymorphism (maybe similar to Haskell type classes) for performance reasons. Integration with an existing IDE (such as MonoDevelop or Visual Studio) is an important addition to a modern language. Finally, addressing the problem of generating less garbage (especially for the XBox 360 and for other platforms such as Windows Phone 7 and iOS through Mono) is another of our objectives.

As a final remark, some aspects of Casanova (namely scripting) already have been fully implemented, as described in [13].

Bibliography

[1] Casanova project page. casanova.codeplex.com/.

[2] Entertainment software association. http://www.theesa.com.

[3] Inheritance vs aggregation in game objects. http://gamearchitect.net/Articles/GameObjects1.html.

[4] Slow garbage collection on the xbox. http://blogs.msdn.com/b/shawnhar/archive/2007/06/29/how-to-tell-if-your-xbox-garbage-collection-is-too-slow.aspx.

[5] The xna framework. http://msdn.microsoft.com/xna.

[6] Xna spacewar 4. http://create.msdn.com/education/catalog/sample/spacewar.

[7] Apostolos Ampatzoglou and Alexander Chatzigeorgiou. Evaluation of object-oriented design patterns in game development. *Inf. Softw. Technol.*, 49:445–454, May 2007.

[8] Mat Buckland. *Programming Game AI by Example*. Jones & Bartlett Publishers, 1 edition, September 2004.

[9] Conal Elliott and Paul Hudak. Functional reactive animation. *SIGPLAN Not.*, 32:263–273, August 1997.

[10] Eelke Folmer. 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'07, pages 66–73, Berlin, Heidelberg, 2007. Springer-Verlag.

[11] Hector Garcia-molina, Jeffrey D. Ullman, and Jennifer Widom. *Database System Implementation*. 2000.

[12] Donald E. Knuth. *The art of computer programming, volume 1 (3rd ed.): fundamental algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[13] Giuseppe Maggiore and Giulia Costantini. *Friendly F# (fun with game programming)*. Smashwords.

[14] Giuseppe Maggiore, Renzo Orsini, and Michele Bugliesi. Casanova: a declarative language for safe games. Technical Report 2011-7, Ca' Foscari - DAIS, 2011.

[15] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1989.

[16] Philip Wadler. Linear types can change the world! In *PROGRAMMING CONCEPTS AND METHODS*. North, 1990.

[17] Philip Wadler. Comprehending monads. In *Mathematical Structures in Computer Science*, pages 61–78, 1992.

[18] 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 '07, pages 31–42, New York, NY, USA, 2007. ACM.