Casanova: a Language for Computer Games

ABSTRACT

In this paper we present 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].

We believe that there is a lack of game programming languages that are powerful enough to build applications which are “more than toys” while still being simple enough to be effectively usable at low levels of competence, for example by young students.

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 [5].

High quality in games is comprised by two main factors: visual quality and simulation quality. Visual quality in games has made huge leaps forward over the years, and both researchers and industry practitioners have been pushing the boundaries of real-time graphics always closer to photorealism; unfortunately, graphics APIs such as DirectX or OpenGL are complex and unfriendly, as witnessed by their gargantuan documentations. Simulation quality, on the other hand, has not evolved through hardware generations as much as rendering. While modern games offer a visual experience that is incomparably superior to that available in the games of 20 years ago, the complexity of the simulation and interaction is only moderately better. 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. It is not by chance that most games published in the last decade are all variations of a few genres. To manage all this complexity, game developers use a variety of strategies. Object-oriented architectures, external scripting languages, reusable components, reactive programming, visual programming, etc. have all been used with some degree of success for this purpose [6] [7] [8] [9] [4]. Finally, networking is still much of an open problem in game development, as there are very few truly high-level general multiplayer engines ready for integration in one’s game, and the understanding of networked architectures is ad-hoc and developed on a game-by-game basis.

In this paper we will present the Casanova language and framework; we stress this distinction right away: while we have designed it as a game programming language in its own right, our work should also be seen as a pervasive design pattern for making games. Nothing prevents a developer from using Casanova outside of the existing implementations, or simply as a descriptive model to

capture the essence of a game "on paper" without dealing with large amounts of technical details. Casanova is the result of a search, started with [10], for a general-purpose methodology that makes game development easier and quicker, by integrating many of the benefits of the above-mentioned techniques, and by simplifying the implementation of rendering and multiplayer for a modern game. We start by describing current approaches to game development in section 2. We describe the general architecture of a modern computer game in section 3. In section 4 we discuss the architecture, syntax, semantics and optimizations of a Casanova game. In section 5 discuss the current state of the implementations. In section 6 we describe how rendering and networking will be supported.

2. RELATED WORK

The two most common game engine architectures found today in commercial games are object-oriented type hierarchies and component-based systems. In a traditional object-oriented game engine the hierarchy represents the various game objects, which all derive from the general Entity class. Each entity is responsible for updating and drawing 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 animations, movements, physics, etc. Component-based systems are being widely adopted, and they are described in [9]. 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; by doing so they lose the focus on the fact that most entities in a game need to interact with each other (collision detection, AI, etc.), and usually lots of a game complexity comes from defining (and optimizing) such 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 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 such as Unity and Unreal Engine (among many others) or even the educational framework such as Alice [4] have been built to allow programming a game with a mixture of a visual programming language (called *editor*), a scripting language for coding the fine-grained behaviors of the various game entities and a separate shading language for defining visual effects.

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, SQL-style declarative programming and integrated shaders inside the main program. Functional reactive programming (FRP, see [8]) has been studied in the context of functional languages. FRP is a data-flow-oriented approach where value modification is automatically propagated along a dependency graph that represents the computation. While FRP offers a solution to the problem of representing long-running behaviors, it neither addresses the problem of many entities that interact with each other, nor does it address the problem of maintaining the consistency of the game world; moreover, FRP architectures have never been studied in conjunction with rendering systems. SQL-queries for games have been used with a certain success in the SGL language (see [11]). This approach uses a lightweight, statically compiled query engine for defining a game. This query engine is optimized in order to allow the programmer to straightforwardly express aggregations and Cartesian products among game entities (very common operators in games of any genre) without worrying about hand-made optimizations. On the other hand, SGL suffers when it comes to representing long-running behaviors, since it focuses exclusively on defining the tick function. As for FRP, SGL has been studied independently from rendering systems. Finally, languages such as Obsidian [12] try to mitigate the difficulties associated with integrating a shading language with the main language of a game; this makes coordination with shader processing easier, cleaner and safer while retaining all the advantages of specifying shaders by hand.

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.

3. ANATOMY OF A GAME

At the heart of a game is a game loop. The game updates and draws the state of the world very often (at least 30 times per second, but 60 is perceived as "smoother") in order to generate more pictures on screen than the human eye can discern. This means that the update of the game logic and the drawing operations may roughly take between 1/30th and 1/60th of a second to complete. Any slower and the game experience will feel broken and un-immersive to the player.

We might model a game as a `draw-update` loop, following the example of the XNA framework (and most other game utility frameworks such as DXUT or GLUT), in the functional language F# [13] like this:

```
type 's Game =  { World : 's
    Update : 's -> Dt -> Input -> 's
    Draw : 's -> ScreenFrame }

let run_game (g : 's Game) =
  let rec loop (g : 's Game) (t : Time) =
    let t' = GetTime() in
    let dt = t' - t in
    let s' = g.Update s' dt (read_input ()) in
    do g.Draw s';
    do loop { g with World = s' } t'
  in loop g (GetTime ())
```

Of course this abstract definition amounts (to some extent) to a formal model of a game, but it has a very serious shortcoming: it really is not saying much! What we would like to do is to define a model of a game that says more about what happens inside the Update and Draw functions, and which helps us with the various constraints of the game development process.

Let us now consider what we are certain to find in a modern game that is not being specified in the game loop described above: *(i)* the data-type `'s` contains many entities of different types, most of which will be stored in collections (projectiles, tiles, monsters, rooms, etc. are always entities in the plural); *(ii)* each entity is updated once for each tick of

the update function; *(iii)* queries on collections are optimized with various techniques, for example indices to speedup Cartesian products and aggregate operations; *(iv)* some aspects of the game logic (often called "scripts") are described with processes that spawn many ticks of the update function; *(v)* a great deal of processing goes into managing the user input; *(vi)* a game draws its entities to the screen; *(vii)* a game often synchronizes its state across multiple instances of itself running across a network. Indeed, Casanova is designed to offer specific functionality for dealing with each of these aspects; while some of these functionalities are, at the time of writing, well understood and tested, others (rendering and networking in particular) are only sketched.

4. ANATOMY OF A CASANOVA GAME

A Casanova program starts with the definition of the game world, a series of (mutually recursive) type declarations rooted at one type: the `GameState`. The `GameState` represents the current state of the game world, and is responsible for storing all the entities that will be updated, rendered and synchronized across the network.

Casanova does not require the developer to specify an update 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.

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 (see 4.3.4 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 we have built an additional scripting system to specify imperative *processes* with coroutines, which smoothly integrate with rules. We use the monadic system of the F# language (F# monads are also known as *computational expressions*) [13] to implement the coroutines, with which we define the main imperative processes and the input processes that are run interleaved with the update loop [10]. 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 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 [14].

The developer then defines the main script, which is an imperative process that is run harmoniously interleaved with the main loop.

The final part of a Casanova program is the definition of 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.

Rules and scripts may use Haskell style list comprehensions to construct lists by ranging over other lists and applying predicates to the range variables. For example, a list comprehension that increments by one all the even values of a list `l` of integers would be written as:

```
[x + 1 | x <- l, x % 2 = 0]
```

We use list comprehensions because of their similarity with SQL-style queries. This gives us an opportunity to optimize the evaluation of these expressions, sometimes even bringing the complexity of their evaluation from quadratic ($O(n^2)$) to logarithmic ($O(n \log n)$) or sometimes even linear ($O(n)$). Casanova lists support constant-time lookup of elements, so the term list is used similarly to how it is used by default in the C# language rather than in the ML-style of immutable linked lists.

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`.

4.1 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 [15], from which Casanova derives strongly.

4.2 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 [15]. 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 the rule belongs to, 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 MyEntity = {
  Position : var Vector2; Velocity : Vector2 }
with rule Position = fun (state,self,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` cycle.

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

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

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:

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

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:

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

Casanova also offers a series of operators that allow us to combine scripts into more complex shapes. The `(&&)` and `(||)` operators run two scripts respectively in parallel (that is, until they both finish) and concurrently (that is, until the first finishes):

$$\frac{\Gamma \vdash s_1 : Script\ T, s_2 : Script\ T'}{\Gamma \vdash s_1\ \&\&\ s_2 : Script\ (T * T')} \qquad \frac{\Gamma \vdash s_1 : Script\ T, s_2 : Script\ T'}{\Gamma \vdash s_1\ ||\ s_2 : Script\ (T + T')}$$

The `(==>)` operator runs a script until it returns `Some x`; when it does, the second script is run with the `x` as input:

$$\frac{\Gamma \vdash s_1 : Script\ (Option\ T), s_2 : T \to Script\ T'}{\Gamma \vdash s_1 \Rightarrow s_2 : Script\ T'}$$

Finally, the repeat operator runs a script forever:

$$\frac{\Gamma \vdash s : Script\ Unit}{\Gamma \vdash repeat\ s : Script\ Unit}$$

We start running a script until it ends, similarly to how one would start a thread, with the run operator. Scripts may only be run inside other scripts:

$$\frac{\Gamma \vdash s : Script\ Unit}{\Gamma \vdash run\ s : Script\ Unit}$$

Notice that our type system enforces rules to be side-effect free, since they cannot perform assignments or start scripts.

The main script is a simple unit script:

$$state : GameState \vdash main : Script\ Unit$$

Each input script is built from two scripts, the event detection script and the event response script:

$$\frac{state : GameState \vdash event_i : Script(Option\ \alpha) \qquad state : GameState \vdash response_i : \alpha \to Script\ Unit}{\vdash event_i \Rightarrow response_i : Script\ Unit}$$

This particular definition of input as a series of pairs of event and response is very important for networking in Casanova; not only does the input system run the appropriate response whenever an event is fired: events also trigger the synchronization of input actions across the network from the clients to the server, as described in section 6.2.

4.3 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 with no overhead.

4.3.1 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 : rule Vector2 = ... }
```

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.

4.3.2 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.

The `update` function is defined as a polytypic function [16] (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, and it fails if it encounters one.

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.

4.3.3 Scripts

Scripts are compiled into F# monads. For a more comprehensive treatment of this mechanism, see [17]. 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, returns either 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. The remaining combinators for scripts can be easily implemented by explicitly manipulating the constructors of the script type.

The parallel execution combinator runs both input scripts one step each. If both scripts terminate, then it returns their results; otherwise, it yields once (with `Next`) and then repeats:

```
let rec (&&) (s1:Script<'a>) (s2:Script<'b>) : Script<'a*'b> =
  fun () ->
    match s1 (),s2 () with
    | Done x, Done y    -> Done (x,y)
    | Next k1, Next k2  -> Next(k1 && k2)
    | Next k1, Done y   -> Next(k1 && (fun () -> Done y))
    | Done x, Next k2   -> Next((fun () -> Done x) && k2)
```

The concurrent execution script runs a step for both of its input scripts. If any of the scripts has finished then its result is returned wrapped into the `Left` or `Right` constructor; otherwise, the script yields once (with `Next`) and then repeats:

```
let rec (||) (s1:Script<'a>) (s2:Script<'b>) : Script<'a+'b> =
  fun () ->
    match s1 (),s2 () with
    | Done x, _         -> Done(Left x)
    | _, Done y         -> Done(Right y)
    | Next k1, Next k2 -> Next(k1 || k2)
```

The *guard* construct keeps running a script, `c`, until its result is `Some x`: when this happens, `x` is passed to the script `s` and the guard script terminates with `s`:

```
let rec (==>) (c:Script<Option<'a>>) (s:'a->Script<'b>)
  : Script<'b> =
  script{
    let! x = c
    match x with
    | Some a ->
      let! res = s a
      return res
    | None ->
      yield
      return! c ==> s }
```

Finally, the repeat script runs a script `s` indefinitely, yielding once at every iteration of `s`:

```
let rec repeat (s:Script<Unit>) : Script<Unit> =
  script{
    do! s
    yield
    do! repeat s }
```

Additional versions of these constructs, such as parallel and concurrent execution on lists of scripts rather than just on two scripts at a time, can be easily derived from the above implementations.

A (mutable) field of type `list<Script<Unit>>` is added to the F# game state. At each tick of the update function each script step is evaluated, and if the script is not finished then its continuation is kept in the list, otherwise it is removed. The initial value of the list of scripts is contains the main script, plus the various input scripts (which are repeated forever).

4.3.4 Optimization

A great deal of developmente 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 [5]; many games feature lots of searches to compare two collections: collision detection, visibility, interaction, etc. We can see an example of such a query in the case study above when finding the asteroids that collide with a single projectile; 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 solve this query in a much shorter time. If the index is a tree, such as a quad-tree, oc-tree, or k-d-tree, then each lookup in the tree will have cost $O(\log n)$. The resulting complexity for the optimized query becomes $O(n \log n)$. Whenever we encounter a query that performs a Cartesian product with some predicate on the generated pairs, then: *(i)* we add to the game state an index that makes resolution of a superset of this predicate faster; *(ii)* at the beginning of the update function we clear and re-fill the index so that it is up-to-date with the current game state; *(iii)* instead of naïvely computing the original query, we look up the index to reduce the number of elements to which we apply the original predicate, but this time on a smaller number of elements. By default Casanova uses a hash table as an index for all query optimizations. The specifics of this technique are well known from the database literature, and can be found in [18]. 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 then the `Next` collection is cleared and the values of the new collection are added to it. Let us consider a simple example:

```
type R = {
  Xs :  list<int> }
  with rule Xs =(fun state,self,dt) -> [x + 1 | x <- self.Xs]
```

Without optimization, the code above would generate a new list at each tick of the update function with the desired value. With our optimization, the field `Xs` uses the type `ResizeArray` for mutable collections instead of the immutable `List`, and the update code becomes:

```
self._Xs.Next.Clear()
for x in self._Xs.Current do self._Xs.Next.Add (x+1)
```

The new code does not allocate a new list of values at each frame but rather reuses the space allocated for the same collection; moreover, the field `Xs` has now type `Rule<ResizeArray<int>>`.

The final optimization that is performed by Casanova is parallel execution. Since at each iteration of the update function there are no rules that write the memory location, given that each rule reads the `Current` value of the other rules but only writes its own `Next` value, then rules may be evaluated (and their results written to the state) in parallel.

4.4 Case Study

Here we show how to build a simple game in Casanova. The asteroid shooter game is a shooter game where asteroids fall from the top of the screen towards the bottom. The player aims his 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; and *(iii)* the current direction the cannon is aiming at, which is updated via the user input:

```
type GameState = {
  Asteroids : list Asteroid; Projectiles : list Projectile;
  CannonAngle    : float32 }
  with rule Asteroids = fun (state, self, dt) ->
      [a in state.Asteroids | a.Colliders.Length = 0 &&
           a.Y < 100.0f]
  rule Projectiles = fun (state, self, dt) ->
      [p in state.Projectiles | p.Colliders.Length = 0 &&
           p.P.Y > 0.0f]
  rule CannonAngle =
    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 }
```

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. Notice that those projectiles are declared with the `ref` type constructor, meaning that they will not be updated during a tick of the update function since they are just references to projectiles which have been updated elsewhere (namely the main collection that contains all the projectiles in the game state):

```
type Asteroid = {
  X : float32; Y : float32; VelY : float32;
  Colliders  : ref list Projectile }
  with rule Y = fun (state, self, dt) ->
    self.Y + dt * self.VelY
  rule Colliders = fun (state, self, dt) ->
      [p in 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 = {
  P          : Vector2;
  V               : Vector2;
  Colliders  : ref list Asteroid }
  with rule P =
    fun (state, self, dt) -> self.P + dt * self.V
  rule Colliders =
    fun (state, self, dt) ->
      [a in 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 omit the initial state for brevity. The main script generates new random asteroids by waiting a random interval between 1 and 3 seconds, and then creating an asteroid with a random `X` coordinate and a random velocity and then repeating the process:

```
let main = {
  let loop =
    { do! wait (random (1.0,3.0))
      do! state.Asteroids :=
          { X = random_float(0.,100.)
            Y = 0.0
            VelY = random_float(5.0,20.0)
            Colliders = [] } :: state.Asteroids }
  in repeat loop
```

Projectiles are generated by waiting for the user to press the space button; when space is pressed, then a new projectile is created with a velocity that corresponds to the

current aim of the cannon, and then it is added to the game state. To avoid generating one projectile every frame while the space key is pressed, we wait one-fifth of a second right after 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:

```
{ if current_input.Space = Pressed then return Some ()
  else return None } ==>
fun () ->
  { let new_projectiles =
        { Position = Vector2(50.0,100.0);
          Velocity =
            Vector2(cos state.CannonAngle,
                    -(sin state.CannonAngle));
          Colliders = [] }
    in
    do! state.Projectiles :=
          (new_projectile :: state.Projectiles);
    do! wait 0.2f }
```

4.5 Benchmarks

We have 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. The more ticks per second the faster a tick, and the more entities we might add to the game world to make it more teeming with interaction:

Optimization	FPS	% gain
None	0.375	N/A
Fast rule swap	0.387	103%
No realloc for lists	0.387	103%
Async	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 of this sample are those of the RTS game in [19].

5. IMPLEMENTATION

It is important to stress out that while we have designed Casanova as a fully-fledged programming language, 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, of the update and draw functions and of networking code. At the time of writing there are three available 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 (for both collections and single values). While this library does not support automated optimizations, 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 [19]. Moreover, the library takes full advantage of all the development tools built to support F# programmers, such as the Visual Studio and MonoDevelop IDEs and debuggers. This library is currently being used and extended organically in the Galaxy Wars research game [20]. The second implementation, which is still a work in progress, is an F# code generator. This implementation takes as input a state definition, a series of F# quotations (code representations in data) for scripts and rules, and generates via reflection the final code which makes use of the library of the first implementation. The input Casanova program is thus fully written in F#, taking fully advantage of its support tools. This implementation generates the optimized code, but the generated code cannot be debugged or browsed. The final implementation, which is also quite ambitious and which will take longer than the others to be completed or even usable, is the language itself.

6. RENDERING AND NETWORKING

Casanova does not yet offer any support for two extremely important aspects of game development: rendering and networking. Rendering and networking, though, have been planned and are currently under study for insertion into Casanova.

6.1 Rendering

Support for rendering in Casanova will be declarative. Rendering in a game can be split into two kinds of rendering: 2D for the UI and for entire, simpler, games and 3D for the game world. Casanova will support drawing pictures and text in 2D and models in 3D. Rendering will be done by adding to the various game entities a series of renderable data-types that contain the path to the drawable asset and the parameters for drawing this asset. The game state will be traversed by the draw function similarly to what happens with the update function, and whenever a drawable value is encountered then this value will be rendered to the screen. For example, 2D pictures may contain a position (in 3D: X and Y for the position on screen and Z for the distance from the observer), a size and a tint to be applied to the original picture:

```
type Picture = { Path : var string; Position : var Vector3
                 Size : var Vector2; Tint : var Vector4 }
```

Similarly, 3D models may contain the path to the model, plus the world, view and projection standard transform matrices. Modern games invariably feature special effects implemented through shaders. Shaders are custom programs that implement interesting transformation of a model from its original representation of triangles and textures into colored pixels on the screen. The first approach one may try could be to define a new model data-type that supports shading, by adding a `Shader` parameter which represents the path to the shader that will be used for drawing the model. Since shaders are parametric programs, it is fundamental to support setting those parameters from the program to a shader. One way to do so might be the introduction of existential types to be able to specify a map of names and values, which are the shader parameters:

```
type Shader = { Path : string;
                Parameters : var map string (∀'a .'a)}
```

This way a shader would now contain a list of parameters that may assume any type. Unfortunately, such a solution does not shield the developer from various mistakes on the name or type of a parameter. A more elegant solution would be to add the shader code to Casanova itself in the form of a code quotation. This way, instead of specifying a shader as a path to an external file, we could use compile-time information from the shader code into the Casanova program to track the shader shape and parameters. The shader data-type would then become:

```
type Shader 'p = { Code : ShaderCode 'p; Parameters : 'p }
```

In the above definition, `'p` is a type variable, synthesized by the Casanova compiler, which stores a record with all the parameters of the shader. This approach would have two important advantages over the previous approach: *(i)* safety over bugs such as uninitialized parameters; *(ii)* higher load-time performance since shaders would be precompiled together with the Casanova source; *(iii)* and higher runtime-performance since we would avoid string lookups when accessing the shader parameters. It is also interesting to notice that this approach could be used to integrate GPGPU computing into Casanova, so that scripts or rules may use the GPU to perform certain lengthy operations on lists. Finally, additional drawable data-types may be added to implement further useful features, for example shading for text and pictures, or predefined shapes such as rectangles, ellipses, etc., or to allow rendering with additional parameters such as alpha-blending or the stencil buffer turned on.

6.2 Networking

The last feature of Casanova is support for multiplayer games without requiring most of the complex work associated with programming networking applications. The architecture of a multiplayer Casanova game will be client-server. Client-server multiplayer games feature multiple, distributed instances of the game running across a network, where one specific instance (usually the one that started the session) is known as the server (or host) and it is the only instance that performs all the computations over its game state and which maintains the updated game state that acts as a reference for all clients. The clients wait for the server to send them the latest values for the variable fields of the state, while sending to the server their input events so their responses may be applied to the reference state and then sent to all clients. Connections are unreliable, and old data is preferably discarded and ignored in order to ensure fast response times; the entire philosophy behind multiplayer games seems to be that of high performance and *eventual consistency*, rather than correctness at all costs. Unfortunately this approach, when implemented in a straightforward manner, suffers from two major shortcomings: *(i)* extreme lag on the clients that need to send their input to the server and then wait a response before seeing the result of their local interaction and *(ii)* excessive bandwidth usage because of too much data sent to synchronize all the state at each tick of the server loop. Commercial multiplayer games use a variety of optimizations to make client-server architectures more viable. Among these, two particular optimizations are widely employed and we have defined their integration with Casanova: *(i)* client-side prediction, that is clients update the non-critical parts of their game state in order to keep the simulation moving and responsive even before the server sends confirmation of those changes, paired with correction of wrong predictions when updated state information finally arrive from the server and *(ii)* partial updates from the server to reduce the necessary bandwidth by sending to clients only those parts of the state which have changed recently, and changing the frequency of the synchronization of state fields and values depending on their importance. By default Casanova will synchronize all the variables of the state, whenever their value changes, from the server to the clients; also, clients will notify input events to the server rather than apply them locally. The probability of sending a variable will be $\sigma\left(1, \frac{1}{k_1}, \frac{change\_age}{k_2}\right)$, where $\sigma(x, y, \alpha)$ is the sigmoid function which interpolates between x and y as α goes from 0 to 1. The developer of a Casanova game will simply state the parameters k_1 and k_2 for each field of each entity. k_1 (*period*) defines how often the field is updated even when it does not change (to make sure that all clients eventually have a chance to synchronize all the state) while k_2 (*time*) defines the amount of time since a change to a field that the probability of synchronizing that field is increased. By default, assignments done to variables or rule values are not carried out on the client (since assignments only come from synchronizations with the server, otherwise client and server may end up with de-synchronized game states). To allow the client to perform assignments on its state, and thus to make the simulation able to partially progress locally without synchronizing with the server, the developer will use the `predict` attribute on entities and their fields.

For example, consider the entity `Ship` in a strategy game. The ship is created with an owner, and it goes from one planet (`Source`) to another (`Target`). As the ship moves from source to target, it fights its enemies and its strength decreases every time it suffers some damage in battle. To avoid keeping a ship still on the client until the next update comes from the server, given that we know that it will move between source and target, we predict its position. This way even if an update for this ship is late from the server the player on the client will still see the ship moving smoothly on his screen. On the other hand there is no way of correctly predicting the other values, and it is better that the user sees no update than seeing, for example, the strength of the ship decreasing too much because of a wrong local update and then jumping back up because some new data is coming from the server. The period between synchronizations of the position and the strength is 20 networking frames (networking by default runs at 20 ticks per second, so a value of 20 means one synchronization per second):

```
type Ship =
  { synchronize(period = 20, time = inf, predict = true)
     Position : var Vector2
     synchronize(period = 20, time = 0.5, predict = false)
     Strength : var int
     Source   : Planet
     Target   : Planet
     Owner    : Player }
```

When no parameters are specified, then default values are picked; preliminary experiments show good results with defaults of: `period = 60`, `time = 2.0`, and `predict = false`.

The techniques described above may work well for various types of games, since they: *(i)* allow for a reduction in the bandwidth used by synchronizing with the right frequency only those attributes that vary; *(ii)* allow for a responsive client-side experience even in the presence of severe lag, since the client may predict certain values instead of needing to wait for the round-trip of telling the server about the modification and then waiting for the server to send it back. Unfortunately, games with a “twitch gameplay” have further needs; “twitch” is the kind of gameplay where the player reaction times are of paramount importance, for example shooter games. In such games, the time it takes for the user input to travel back and forth to the server to

signal that a player is shooting may be too long, to the point that the target may have already moved away from the line of fire. This kind of game requires techniques for making certain computations directly on the client, thereby introducing small, local de-synchronizations which the server then receives and integrates into the main state. These techniques usually require to interpolate or extrapolate (through the technique known as “dead reckoning”) the game state locally on the client, and to compensate for lag by carefully validating client-side events on the server. We are currently studying how to cleanly integrate the above techniques, for example as presented in [21].

7. CONCLUSIONS AND FUTURE WORK

Games and multimedia applications are extremely widespread. Disciplined models and techniques that simplify game development have a unique chance of having a significant impact. 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, 3D graphics and even multiplayer.

The Casanova implementation is proceeding steadily and Casanova can be used already, albeit with some limitations (rendering and networking implementation is still at an early stage). Our aim is to make Casanova into a *complete* solution for game development. In the short term we are planning on adding rendering and networking. In the long term, we are studying how to integrate menus, sounds, procedural generation of content, AI techniques, and GPGPU through the same shading mechanism used for rendering.

By using and evolving Casanova in non-trivial, sample projects such as [20] we are empirically measuring a great increase in both productivity and enjoyment for developers, since programmers have a chance to focus more on coding features than on fixing bugs. We expect that with further work the importance and usefulness of such a framework will become even more evident.

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

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

7 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.

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

9 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.

10 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.

11 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.

12 Svensson, Koen Claessen and Mary Sheeran and Joel. Obsidian: GPU programming in haskell. (International Symposium on the Implementation and Application of Functional Languages (IFL '08) 2008).

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

14 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.

15 Pierce, Benjamin. *Types and Programming Languages*. MIT Press, Cambridge, Massachusetts , 2002.

16 Jeuring, Patrik Jansson and Johan. PolyP - a polytypic programming language extension. (1997), Symposium on Principles of Programming Languages (POPL).

17 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).

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

19 Maggiore, Giuseppe. Casanova project page. In *http://casanova.codeplex.com/* (2011).

20 Maggiore, Giuseppe. Galaxy Wars Project Page. In *http://vsteam2010.codeplex.com* (2010).

21 Bernier, Yahn. Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization. *Valve Corporation* (2001).

22 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.

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

24 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.

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

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

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

28 Wadler, Philip. Comprehending monads. In *Mathematical Structures in Computer Science* (1992), 61–78.