Casanova: A simple, high-performance language for game development

Mohamed Abbadi, Francesco Di Giacomo, Agostino Cortesi
Pieter Spronck, Giulia Costantini, Giuseppe Maggiore

Università Ca' Foscari DAIS, Venice, Italy
Tilburg University, Tilburg, Netherlands
Hogeschool Rotterdam, Rotterdam, Netherlands
`{mohamed.abbadi,francesco.digiacomo,cortesi}@unive.it`
`p.spronck@uvt.nl`
`{costg,maggg}@hr.nl`
`http://casanova.codeplex.com/`

Abstract. Managing the flow of time and the coordination of multiple components in games (and other highly interactive applications) is a challenging task. Therefore game development requires a lot of effort, even for (apparently) simple scenarios. To reduce the cost and effort of game development, we designed a new computer language called "Casanova 2". Using a case study, we demonstrate that Casanova 2 can be used to implement typical game scenario's using functional programming constructs. Our evaluation shows that it has both a high performance and a high usability.

Keywords: Game development, Casanova 2, languages, functional programming

1 Introduction

Computer and video games are a big business, and they have grown to the point that their sales are higher than those of music and movies [5]. For mobile games alone, predicted sales for 2017 exceed 100 billions of dollars [20] . The relevance of games as a social phenomenon is at an all-times high.

In the wake of this widespread adoption of games, their application areas have expanded. They are no longer used exclusively for entertainment. They have found applications in education, training, research, social interaction, and even raising awareness [7, 19]. These so-called *serious games* are used in schools, hospitals, industries, and by the military and the government. Researchers use games to simulate environments and evaluate their results. These games do not enjoy the same rich market of entertainment games, but their social impact is nevertheless high [13].

An obstacle to the widespread use of serious games is that they are expensive to build. Since the resources of those who want to use serious games are usually quite limited, many serious games (and game-based research projects) fall short of their technological mark or fail altogether. This constitutes a high threshold that may cause innovative projects to be shut down prematurely.

The expenses of building a game are tightly related to the complexity of the structure of the game itself. At its core, a game features a state which describes the game world, plus the game loop that describes how the state changes over time. To avoid duplication of substantial effort in these areas, the game industry often uses ready-made components (also called "engines") [8], for example for graphics and physics, which *encapsulate* functionality in a modular way. As the number of components increases, complex concurrent [6] interactions between the components need to be realized. This yields additional difficulty.

Our main goal in this paper is to reduce the complexity of game code. We introduce *proper abstractions for game development, and build programming tools that implement those abstractions.* These abstractions would help developers in reaching their goals by substantially reducing development efforts, with a special benefit for smaller development teams that work on serious games. Since games are interactive applications, the proposed abstractions should not compromise the performance of the programs.

We begin with a discussion of games and their complexity, introducing a case study. We use our case study to identify issues in the way games are traditionally expressed (Section 2). We propose a tiny, concurrency-oriented, game-centered language (Casanova 2) for describing game logic, and show how the case study is expressed in this language (Section 3). We then evaluate the effectiveness of Casanova 2 for creating games in terms of performance and simplicity (Section 4). Sections 5 and 6 cover future work and conclusions.

2 Technical challenges in games development

In this section we discuss games and their complexity through a case study. We consider a sample which fits within the space constraints of a paper, which at the same time shows the complex interactions that are typical for games.

2.1 Running example

The running example we use is a patrol moving through checkpoints. The state of the patrol is made up by the position of the patrol `P`, and its velocity `V`.

```
P is a 2D Vector
V is a 2D Vector
Checkpoints is a list of 2D Vectors
```

The logic of the game is given using a pseudo language:

```
P is integrated by V over dt
V points towards the next checkpoint until
  the checkpoint is reached, then becomes
  zero for ten seconds (the patrol is idle)
```

A game is said to run as a sequence of time slices, called "frames." A typical game runs at 30 to 60 frames per second. The pseudo code above describes the logic of the patrol, which runs every frame. The logic shows a typical dynamic present in any game, which is made up by continuous components (the update of `P` in our case) and discrete components (the update of `V`). As a result, `P` changes every frame, while `V` only changes upon reaching a checkpoint.

2.2 Common issues

Dynamics such as the one described above are built in games either with engines or by hand.

Engines An engine is a library built to offer solutions for specific tasks (such as graphics and physics control) in order to speed up the development process, by promoting code reuse and reducing mistakes. Examples of commonly used engines are: MissionMaker [2] and GameMaker [1]. By hiding complexity inside libraries and editors, developers only need to adapt their design to the engine. While popular, engines tend to have significant issues:

- Engines are often difficult, or even impossible, to expand and to adapt to the needs of the developer using them; this limits developers because some aspects of design might need to be adapted or left out due to of lack of specific support by the engine;
- Engines are often closed libraries. Even though engines are internally optimized, the possibilities for global optimization that take into account the game structure are very limited;
- Expertise is needed to master an engine. Since most (if not all) engines are highly complex to use, a significant effort of the developers is spent on learning the intricacies of the engine.

In general, a good engine offers good performance and a reduced possibility to make mistakes, but at the same time limits developers to the engine features and asks them to master most features before using the full capabilities of the engine.

Hand made implementations A hand made implementation is used when developers are looking for specific behaviors, want to have more control on the game implementation, or when the support of the underlying platform is poor. Hand made implementations raise important issues to be considered before starting a new project:

- Games tend to be very large applications. As size increases, the number of interactions increases as well, together with the possibility to make mistakes;
- Optimization (when done by hand) adds complexity, because it requires supplementary data structures and may change the implementation of the game interaction. Optimization may also lead to (*i*) implementation issues (for instance some optimization may work only on specific architectures), and (*ii*) maintainability issues (any change in the game design should keep into account its repercussions on the implementation).

We now present an example of hand-made implementation of the patrolling dynamics following the style of [14]:

```
class Patrol:
  enum State:
   MOVING
   STOP

  public P, V, Checkpoints
  private myState, currentCheckpoint, timeLeft

  def loop(dt):
   P = P + V * dt
   if myState == MOVING:
     if P == Checkpoints[currentCheckpoint]:
       myState = STOP
       V = Vector2.Zero
       timeLeft = 10
   elif myState == STOP:
     if timeLeft < 0:
       currentCheckpoint += 1
       currentCheckpoint %= Checkpoints.length
       myState = MOVING
       V = Normalize(
            Checkpoints[currentCheckpoint] - P))
     else
       timeLeft -= dt
```

The `loop` function implements the patrolling behavior. It takes one argument `dt` which represents the delta time elapsed since the last frame. The very first line of the `loop` body implements the position update behavior. The velocity behavior depends on whether the patrol is moving or idle. While moving, we stop the patrol as soon as he reaches the checkpoint, and set the wait timer to 10 seconds. If the patrol is idle and the countdown is elapsed, the next checkpoint is selected. At this point the patrol points toward the new checkpoint and starts moving again.

2.3 Discussion

The patrolling sample illustrates a common division between design and implementation in games. Deceptively simple problem descriptions turn out to require surprisingly articulated implementations. Complexity mainly originates from the explicit definition and management of a series of *spurious variables* that are needed to program the logical flow of the problem but which do not come up in the design. In our case study, the spurious variables are `myState` (together with the definition of the state structure) and `timeLeft`.

A language suited for game development by persons for whom game development is not their main job, has two main requirements:

– *Performance*: games are highly interactive applications which tend to be filled with many dynamic elements; if the language in which they are built does not guarantee high performance, the player experience will suffer;
– *Simplicity*: the language should be easy to understand and easy to express game functionality in; if the language is not simple in these respects, it

requires an amount of training that is not within reach of those who are not game developers by profession.

Below we introduce a game-centered programming language and show how to rebuild the sample above with fewer spurious constructs, in a way that is closer to a higher-level, readable description.

3 Casanova 2

Languages, in general, offer more expressive power than engines, because of the possibility to combine and nest the constructs. A language specifically designed and built with game programming in mind can help with common aspects of game development (such as time, concurrency, and state updates) that regular languages do not encompass. In this regard, we present the language Casanova 2, based on [11], which takes its inspiration from the orchestration model of [15]. We show how Casanova 2 is designed in particular to express the typical dynamics present in games.

3.1 The basic idea behind Casanova 2

An abstraction of a game should be able to represent its main elements, i.e., its state variables and their (discrete and dynamic) interactions. For this purpose, we built an (intentionally) small programming language of which the main features are *state* and *rules*:

(i) The *state* of a game is represented by a hierarchical type definition. Each node of the hierarchy is called an *entity* (besides the root, which is called *world*). Each entity contains a series of fields that represent primitive types, collections, or even references to other entities. Through access to shared data entities we achieve concurrent coordination.

(ii) The logic of each entity is defined as a series of implicitly parallel looping code blocks. Each implicit block, called a *rule*, represents a specific dynamic of the entity. A rule represents a dynamic, which can be continuous (simple and effect-free) or discrete (with side-effect, the most important of which is *wait*).

3.2 Casanova patrol

We now show how we rewrite the patrol program presented in Section 2 using Casanova 2.

Listing 1.1. Patrol in Casanova 2

```
world Patrol = {
   V : Vector2
   P : Vector2
   Checkpoints : [Vector2]

   rule P = P + V * dt
```

```
    rule V =
     for checkpoint in Checkpoints do
       yield ||checkpoint - P||
       wait P = checkpoint
       yield Vector2.Zero
       wait 10<s>
}
```

The first three lines within the definition of Patrol describe the game state, containing three variables: the velocity `V`, the position `P`, and a checkpoint list `Checkpoints`. The next line gives the continuous dynamic, namely the rule P which runs once per frame, i.e., at every frame the position `P` is integrated by the velocity `V` over `dt` (a global value supplied by the system that represents the time difference between the current and the previous frame). The remainder of the definition gives the discrete dynamic, namely the rule V, which represents the movement between checkpoints. The checkpoints are traversed in order, and for each selected checkpoint `checkpoint` we change the value of the velocity in order to move the patrol towards it (`yield checkpoint - P`). Then, we wait until the patrol reaches the checkpoint (`wait P = checkpoint`), and once the checkpoint is reached we stop the patrol, by setting its velocity to 0 (`yield Vector2.Zero`) for 10 seconds (`wait 10<s>`). At this point the loop continues and a new checkpoint is selected. We reiterate the list again once we have traversed all the checkpoints.

Note that, in general, a game can be considered a series of entities that run in synchronization in order to achieve a specific goal. In Casanova 2 every entity in the state (as well as every rule in an entity) is in essence an *independent* concurrent program [17]. Coordination between these programs happens through a shared state.

3.3 Syntax

The syntax of the language (here presented in Backus-Naur form [18]) is rather brief. It allows the declaration of entities as simple functional types (records, tuples, lists, or unions). Records may have fields. Rules contain expressions which have the typical shape of functional expressions, augmented with `wait`, `yield`, and queries on lists:

Listing 1.2. Casanova 2 syntax

```
<Program> ::=
    <moduleStatement> {<openStatement>}
    <worldDecl> {<entityDecl>}

<moduleStatement> ::= module id
<openStatemnt>    ::= open id
<worldDecl>       ::= world id ["("<formals>")"] =
                      <worldOrEntityDecl>
<entityDecl>      ::= entity id ["("<formals>")"] =
                      <worldOrEntityDecl>
<worldOrEntityDecl> ::= "{" <entityBlock> "}"
<entityBlock>  ::= {<fieldDecl>} {<ruleDecl>}
                   <create>
<create> ::= Create "(" {<formals>} ") = <expr>
```

```
<formals>   ::= id [":" <type>] {"," <formals>}
<fieldDecl> ::= id [":" <type>]
<ruleDecl>  ::= rule id {"," id} "=" <expr>
<type>      ::= int |boolean  |float |Vector2
                 |Vector3 |string |char
                 |list "<" <type> ">" |<generic>
                |<type> "[" "]" |id
<generic>     ::= "'" id
<expr> ::= ...(* typical expressions : let, if,
                 for , while , new, etc. *)
           | wait (<arithExpr> | <boolExpr>)
           | yield | <arithExpr> | <boolExpr>
           | <literal> | <queryExpr> | <seq>
<seq>         ::= <expr> <expr>
<arithExpr>  ::= ...//arithmetic expressions
<boolExpr>   ::= ...//boolean expressions
<literal>    ::= ...//strings , numbers
<queryExpr>  ::= ...//query expressions
```

3.4 Semantics

The semantics of Casanova 2 is *rewrite-based* [10], meaning that the current game world is transformed into another one with different values for its fields and different expressions for its rules. Given a game world ω, the world is structured as a tree of entities. Each entity E has some fields $f_1 \dots f_n$ and some rules $r_1 \dots r_m$.

```
E = { Field$_1$ = f$_1$; ...; Field$_n$ = f$_n$;
      Rule$_1$ = r$_1$; ...; Rule$_m$ = r$_m$ }
```

Each rule acts on a subset of the fields of the entity by defining their new value after one (or more) ticks of the simulation. For simplicity, in the following we assume that each rule updates all fields simultaneously.

An entity is updated by evaluating, in order, all the rules for the fields:

```
tick(e:E, dt) =
 { Field$_1$=tick(f$_1^m$, dt); ...; Field$_n$=tick(f$_n^m$, dt);
   Rule$_1$=r$'_1$; ...; Rule$_m$=r$'_m$ }
where
  f$_1^m$, ..., f$_n^m$, r$'_m$ = step(f$_1^{m-1}$, ..., f$_n^{m-1}$, r$_m$)
  .
  .
  f$_1^1$, ..., f$_n^1$, r$'_1$ = step(f$_1$, ..., f$_n$, r$_1$)
```

We define the `step` function as a function that recursively evaluates the body of a rule. The function evaluates expressions in sequential order until it encounters either a `wait` or a `yield` statement. It also returns *the remainder of the rule body*, so that the rule will effectively be resumed where it left off at the next evaluation of `step`:

```
step(f$_1$, ..., f$_n$, {let x = y in r$'$}) =
  step(f$_1$, ..., f$_n$, r$'$[x:=y])

step(f$_1$, ..., f$_n$, {if x then r$'$ else r$''$; r$'''$})
  when (x = true) = step(f$_1$, ..., f$_n$, {r$'$; r$'''$})

step(f$_1$, ..., f$_n$, {if x then r$'$ else r$''$; r$'''$})
  when (x = false) = step(f$_1$, ..., f$_n$, {r$''$; r$'''$})
```

```
step(f1, ..., fn, {yield x; r'}) = x, r'

step(f1, ..., fn, {wait n; r'})
  when (n > 0.0) = f1, ..., fn, {wait (n-dt); r'}

step(f1, ..., fn, {wait n; r'})
  when (n = 0.0) = step(f1, ..., fn, r')

step(f1, ..., fn, {for x in y:ys do r'; r''})
  step(f1, ..., fn,
       {r'[x:=y];
        for x in ys do r'; r''})

step(f1, ..., fn, {for x in [] do r'; r''})
  step(f1, ..., fn, r'')
```

3.5 Compiler description

Specific syntax built around the concept of altering the execution flow of a Casanova program allows the Casanova compiler to translate a Casanova program into an equivalent and high performance low-level program with the same semantics. The result is a high performance program made by a single switch structure, without nesting. A big advantage of this solution is that we may ignore typical software engineering rules, such as readability and code maintainability (as readability and maintainability are only needed for the Casanova specification of the game).

Usually, software engineering implementations are based on a series of nested state machines, but nesting yields a low performance because of the state selection. In contrast, the Casanova compiler produces an inlining of all the nested state machines into a single sound and fast state machine (which code is pretty much unreadable).

4 Evaluation

In this section we present a comparison between Casanova and other programming languages used for game development. The evaluation is based the on two essential aspects mentioned in Section 2: *performance* and *simplicity*. Performance is a fundamental indicator of the feasibility of a programming language that needs to be used in a resource-conscious scenario such as games. Simplicity is important as well, especially in those scenarios where development time and expense constitute a major concern (like for serious and indie game studios).

In particular we observe that, in many cases, programming languages for games offer a difficult either-or choice between simplicity and performance. As we will show, Casanova solves this apparent dichotomy by offering both at the same time.

4.1 Tested languages

We have chosen four languages which represent various development styles and which are all used in practice for building games. We have mostly focused on

Table 1. Performance comparison

Language	Time per frame
Casanova	0.07ms
C#	0.12ms
JavaScript	24.07ms
Lua	20.90ms
Python	20.15ms

those languages which are used for building game logic, and we have shied away from considering languages (such as C++) which are used for building engines or libraries [8], as Casanova is not "competing" with them. Three of the chosen languages are dynamically typed programming languages: Lua, JavaScript, and Python, which have as their main selling points simplicity and immediacy [9]. The fourth chosen language is C# because of its good performance and relative simplicity. One could argue that we are comparing our language, which might appear as a Domain Specific Language (DSL), with General Purpose Languages (GPL's). However, Casanova 2 is actually a GPL, although its main field of application is computer game development.

The "benchmark sample" simulates a game with ten thousands patrols. We made an effort towards implementing the sample by using coroutines and generators [12] whenever available, in order to express the game logic in an idiomatic style for each language. In order to compare the language functionality, we are only running the logic of the game and we do not execute any other component unrelated to it (such as the graphics engine), to produce a fair performance benchmark. The code samples can be found on [3].

4.2 Performance

We have generated tens of thousands of entities in a loop that simulated a hundred thousand frames. This corresponds roughly to half an hour of play time on a reasonably crowded scene. The results are summarized in Table 1.

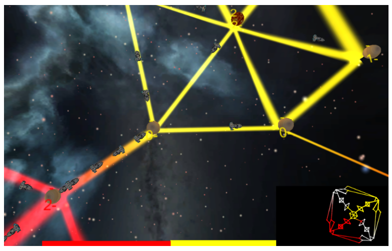
As we can see from the table, the performance of Casanova 2 is of the same order as C#, and is multiple orders of magnitude faster than that of the scripting languages. In this simple but populated scenario, the limits of Lua, Python, and JavaScript, deriving from the high cost of dynamic lookup, are clearly shown. In addition, all languages use virtual calls to methods, such as those for managing and executing coroutines and generators, which add overhead at the expense of performance. In short, Casanova 2 generates highly optimized rule code which does not require general purpose constructs, such as coroutines in games, that often use virtual methods and dynamic lookups. In a sense, Casanova 2 uses all the static information it can to avoid work at runtime.

We believe that it is worthy of notice how much Casanova in this prototypical implementation offers a performance which is even better than that of a very high quality and mature compiler such as that of C#.

Table 2. Syntax comparison

Language	Syntagms	Lines of code	Total words
Casanova	47	31	104
C#	61	69	269
JavaScript	52	41	257
Lua	47	45	249
Python	50	34	214

(a) Dyslexia



(b) RTS

Fig. 1. Casanova games

4.3 Ease of use

Assessing the simplicity of a programming language is a daunting task. Just as much as beauty lies in the eye of the beholder, simplicity in programming languages heavily depends on the programmer preferences, history, and previous knowledge.

For the comparison we have taken the benchmark sample and implemented it with each of the considered languages. In order to assess the complexity of the sources we have:

1. counted the number of lines of each source, thereby assessing the size of the implementation, with the assumption that a bigger sample corresponds to more complexity;
2. counted the number of keywords and operators (syntagms) that come into play for each implementation, with the assumption that a high count corresponds to more required knowledge from the developer.

The results are summarized in Table 2. As we can see, Casanova 2 resulted in significantly less lines of code and syntagms, especially with respect to C# (the only other language with comparable high performance).

4.4 Summary

In conclusion, Casanova 2 has been shown to offer both good performance and simple code at the same time. On the one hand, Casanova 2 is as simple as "easy to use" scripting languages. On the other hand, this simplicity does not come with the usual associated hit in performance that characterizes these languages. The performance of Casanova is even a bit better than that of C#, a highly optimized commercial language. A series of applications has been built with the language as part of teaching and research projects. One of those is an RTS game (see Figure 1b) that features complex integration with a professional-quality engine, Unity3D[4]. The other notable application is a game for detecting dyslexia in children (see Figure 1a). The game is currently being used as a tool for research and features some articulated animations and state machines. These applications can be found at `http://casanova.codeplex.com/`.

5 Future work

The Casanova 2 language is capable of implementing usable and quite complex games. The language, while usable, is currently still in development as it misses a few features. In particular, support for multiplayer games is at this moment lacking. We believe that the existing mechanisms for handling time offered by Casanova 2 could be augmented with relatively little effort in order to greatly simplify the hard task of building multiplayer games. This is part of future work, that we are currently engaging in. We are also doing usability studies using students from various disciplines and backgrounds.

The high level view of the game that the Casanova 2 compiler provides can be exploited in order to improve the programmer experience. This means that we could use tools for code analysis (such as abstract interpretation [16] or type system extensions) in order to better understand the game being built, and to help with correctness analysis, performance analysis, or even optimization.

6 Conclusions

Casanova 2, a language specifically designed for building computer games, may offer a solution for the high development costs of games. The goal of Casanova 2 is to reduce the effort and complexities associated with building games. Casanova 2 manages the game world through entities and rules, and offers constructs (wait and yield) to deal with the run-time dynamics. As shown by the benchmarks in Section 4, we believe that we have taken a significant step towards reaching these goals. In fact, we achieved at the same time very good performance and simplicity, thereby empowering developers with limited resources.

References

1. Gamemaker. `http://www.immersiveeducation.eu/index.php/missionmakerm`.
2. Missionmaker. `https://www.yoyogames.com/studio`.
3. Performance evaluation code comparison. `https://casanova.codeplex.com/wikipage?title=Casanova%20Performance%20Comparison`.

4. Unity 3d. `https://unity3d.com/`.
5. Essential facts about the computer and video game industry 2011, 2011.
6. S. Bilas. A data-driven game object system. In *Game Developers Conference Proceedings*, 2002.
7. I. Bogost. *Persuasive games: The expressive power of videogames.* Mit Press, 2007.
8. J. Gregory. *Game engine architecture.* CRC Press, 2009.
9. T. Gutschmidt. *Game Programming with Python, Lua, and Ruby.* Premier Press, 2004.
10. J. W. Klop and R. De Vrijer. *Term rewriting systems.* Centrum voor Wiskunde en Informatica, 1990.
11. G. Maggiore, A. Spanò, R. Orsini, G. Costantini, M. Bugliesi, and M. Abbadi. Designing casanova: a language for games. In *Advances in Computer Games*, pages 320–332. Springer, 2012.
12. C. D. Marlin. *Coroutines: A programming methodology, a language design and an implementation.* Number 95. Springer, 1980.
13. D. R. Michael and S. L. Chen. *Serious games: Games that educate, train, and inform.* Muska & Lipman/Premier-Trade, 2005.
14. I. Millington and J. Funge. *Artificial intelligence for games.* CRC Press, 2009.
15. J. Misra and W. R. Cook. Computation orchestration. *Software & Systems Modeling*, 6(1):83–110, 2007.
16. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis.* Springer, 1999.
17. F. B. Schneider. *On concurrent programming.* Springer, 1997.
18. L. Strings. Backus-naur form. *Formal Languages syntax and semantics Backus-Naur Form 2 Strings, Lists, and Tuples composite data types*, 2010.
19. T. Susi, M. Johannesson, and P. Backlund. Serious games: An overview. 2007.
20. M. Tim. Global games investment review 2014. `http://www.digi-capital.com/reports/`, 2014.