High performance encapsulation and networking in Casanova 2

Abstract

Encapsulation is a programming technique that helps developers keeping code readable and maintainable. However, encapsulation in modern object-oriented languages often causes significant runtime overhead. Developers must choose between clean encapsulated code or fast code. In the application domain of computer games, speed of execution is of utmost importance, which means that the choice between clean and fast usually is decided in favor of the latter. In this paper we discuss how encapsulation is embedded in the Casanova 2 game development language, and show how Casanova 2 allows developers to write encapsulated game code, which thanks to extensive optimization achieves at the same time high levels of performance. Furthermore, we show that the abstractions provided by Casanova so far cover no more than the tip of the iceberg: we document a further extension in the traditionally challenging domain of networking and show how the language can provide significant improvement in productivity.

Keywords: Domain Specific Language, Encapsulation, Networking, Games

1. Introduction

The video games industry is an ever growing sector with sales surpassing 20 billion dollars in 2014 [1]. Video games are not only built for entertainment purposes, but they are also used for Edutainment, Higher Education, Health Care, Corporate, Military, Research, and others [2, 3]. These so-called *serious games* usually do not enjoy the budgets available in the entertainment industry [4]. Therefore, developers of serious games are interested in tools capable of overcoming the coding difficulties associated with the complexity of games, and reducing the long development times.

Video games are composed of several inter-operating components, which accomplish different and coordinated tasks, such as drawing game objects,

running the physics simulation of bodies, and moving non-playable characters using artificial intelligence. These components are periodically activated in turn to update the game state and draw the scene. When game complexity increases, this leads to an increase in size and complexity of components, which, in turn, leads to an increase in the complexity of developing and maintaining them, and thus an increase of development costs.

A possible approach to reduce development costs is to use game development tools (e.g., GameMaker, Unity3D, or UnrealEngine [5]), which tend to produce simple games that are hard to customize and bound to a specific genre. To provide some level of customization, game developers rely on general-purpose languages (GPLs) [6]. GPLs are typically unable to provide domain-specific abstractions and constructs. This means that when developing games by means of a GPL the resulting code will be complex and expensive to maintain [7, 8]. According to [9], the typical life cycle of software implemented by means of a GPL is: (***i***) building a prototype; (***ii***) designing a version in which the code is readable and maintainable; and eventually (***iii***) refactoring, after obtaining confidence with the context and the problem, the code from the previous point, so as to realize the last (often non-functional) requirements.

We can see that the just introduced cycle is applicable to game development as well: (***i***) building a game prototype is always necessary to take confidence with the context of the problem and the chosen tool; (***ii***) designing game code that is maintainable and readable requires developers to abstract the problem and to focus more on the high-level interactions of the game and its data structures. Software development techniques have been studied to improve software maintainability and tackle complexity [10]. Encapsulation, which consists of isolating a set of data and operations on those data within a module and providing precise specifications for the module [11], is an example of a technique aimed at increasing code maintainability and readability; and (***iii***) refactoring is a common process in game development, see for example the case of performance optimization, which is of high importance for games, since it is strictly connected to game smoothness, i.e., to the game's frame rate, and smoothness strongly influences the perceived quality of a game [12]. Indeed developing a game is a highly dynamic process [13] involving a wide variety of team members with different roles, such as designers, programmers, etc. Design very often changes during the development stage, as proven in several examples from the industry, such as Starcraft, Duke Nuke'em Forever, and Final Fantasy XV [14]. Small

changes at the abstract design level translates into considerable amount of code, which might affect the overall architecture; thus every stage of the above life cycle requires effort and is time consuming. An example of this is when using encapsulation [15] in the code. Since a game may feature many small entities, encapsulation forces those entities to interact through specific interfaces. When calling methods of the interfaces, overhead is added due to dynamic dispatching [15]. Such overhead ultimately affects the performance of games at runtime negatively, so a complete refactoring that accommodates performance becomes necessary. Similar negative effects come from various design patterns, which all add layers of indirection. These effects impact negatively cache coherency and force CPU prediction failures. Traditional networking in games is another example that typically breaks encapsulation as the what to send over the network is dependent on the game logic; thus small changes in the game structure could affect heavily the networking layer.

What seems ideal is to have the advantages coming from both stages (*ii*) and (*iii*): game code that is well maintainable and readable, and at the same time with a fast run time. For this purpose, we investigated this problem and developed a solution that allows developers to write encapsulated code. Encapsulation is "a language mechanism for restricting direct access to some of the object components.". According to the definition of encapsulation, data and operations on them must be isolated within a module and a precise interface must be provided. Our solution turns, through extensive automated optimization, encapsulated code into an equivalent high-performance executable, therefore relieving developers from refactoring important design structures by hand, thus reducing the chances to make mistakes. As a further note, we want to underline that this optimization could be performed at source code level; however the logic of the program would be irremediably lost in the complex details of the architecture needed for the optimization, and thus debugging the code would be impractical. This is why we decided to look for a solution at a lower level of abstraction.

To sum up, in this paper we present a solution, which makes use of optimization transformations, that addresses the problem of the loss of performance in encapsulated games and of abstracting networking primitives. We present our solution as an extension for a domain specific language for games, called "Casanova 2", which allows developers to write high quality games at reduced development costs.

We start with a discussion about the focus of this paper[1] and related work (Section 2). Then we start with a discussion of encapsulation and typical optimizations (which break encapsulation) and their complexity, by introducing a case study. We use the case study to identify issues in using both encapsulation and faster implementation for games (Section 3). We introduce our idea for dealing with encapsulation without losing performance (Section 4). We propose a specific implementation, with corresponding semantics, within the Casanova 2 language (Section 5). We discuss a further extension of Casanova 2 in the domain of networking and show how the language can also provide significant improvement in productivity (Sections 6 and 7). We then evaluate the effectiveness of our approach in terms of performance and compactness (Section 8), round off with conclusions, and present future challenges within this scope (Section 9).

2. Focus of the work and related work

The focus of this paper lies exclusively within the restricted, non-general-purpose field of game development (and its sibling, real-time simulations). This greatly narrows the scope of the problem, but also severly constrains the spectrum of possible solutions. To understand this, consider that on one hand we have the deep complexity of the underlying mathematics of the physical aspects of the game and the highly concurrent nature of the discrete logic; on the other hand, we have the fundamental, pervasive non-functional requirement that no single update/draw cycle may ever take more than 1/60th of a second in total. Whereas in other soft-real-time domains, one might occasionally accept a degradation of performance, provided that the variance of the distribution of computational cycles is acceptably low, the game becomes a clear failure if any frame is delayed.

This very strict performance requirement automatically excludes a large number of (admittedly beautiful and powerful) frameworks that in and of themselves would solve many architectural issues that games do need to face. This brings us to try to address the focus of the paper without tackling the general issue. We do believe that tackling the general issue of separation of concerns and real-time performance as required for games is still outside of

[1]This paper is an extended version of a conference paper that appeared as [16]. The key additions of this journal version are: an extended related work session, and a new section on networking within the Casanova language.

the boundaries of what can be achieved with modern tools and as such limited work like the present paper explores an interesting direction of investigation.

The general-purpose frameworks that might be used in our present context can be classified in two broad areas: runtime dynamic machinery, and compile-time code generators.

2.1. Runtime dynamic machinery

Highly dynamic frameworks typically make use of mechanisms that either feature large amounts of dynamic/virtual calls, or rely on reflection. The use of dynamic/virtual calls within a big hierarchy of objects has dramatic effect on performance [17] because it severely disrupts cache coherency. This is unfortunate, as it rules out the widespread use of design patterns such as decorators, and in the functional programming world the extensive use of monads.

Reflection mechanisms (for example reflection in .NET [18]) tend to be even less effective than mechanisms with large amounts of dynamic/virtual calls, as they combine the same number of cache disruptions with the need to box/unbox everything and constantly check for the correct types of boxed arguments. Among the frameworks that use this technique, we find (*i*) Proxies in C#, an aspect-oriented library supported by the .NET framework, and (*ii*) netty.io, an event-driven framework for networking. The overhead of these techniques makes it unfortunately very easy to exceed the maximum allotted time of 1/60th of a second per frame, or requires to dramatically reduce the number of entities processed by the game, which in turn results in a poorer game experience.

2.2. Compile-time code generators

A more promising venue of investigation is that of compile-time code generators, which make it possible to implement sophisticated, reusable meta-patterns such as those discussed above, but without having to rely on expensive forms of dynamism. Examples of such generators are Haskell templates, C++ templates, and macros in Lisp. The performance of these generators is clearly bound to the performance of the underlying language. As we already discussed, performance is a very strict and stringent requirement within our domain of focus, and so this immediately excludes frameworks based on languages such as Haskell or Java that have less control on performance because of large amounts of boxing (in Haskell laziness induces boxing). Other

frameworks offer less disciplined meta-structures. For example, C++ templates lack a higher-kinded type system that would allow us to constrain type parameters and get some measure of control on error messages. While this might seem trivial, C++ templates are very unwieldy to use and debug because the untyped replacement mechanism generates pages of errors in otherwise correct libraries only because they have been instantiated with the wrong parameters.

Moreover, hybrid frameworks, such as *Treecc* (an Aspect-Oriented approach to writing compilers), force patterns on the generated code which make too much use of polymorphism. This partially defeats the point of compile-time code generators for games, as it still causes performance issues such as those outlined in [17].

Furthermore, meta-programming approaches would bring discipline to our work, which we are actively working on improving (see Section 9).

3. Encapsulation in games

In this section we discuss first common issue arising from traditional facilities for game development. We then introduce a short example to explain the problem of encapsulation in games. Eventually, we discuss the advantages and disadvantages of using encapsulation when designing a game.

3.1. Common issues

In the panorama of game development the two main approaches to game development are either the use of **tools**, or the use of **languages** [19].

Tools are environments where developers are assisted in the creation of games through visual instruments and built-in features (such as physics). Tools are generally focused on specific genres. A typical aspect of these tools is that they offer developers predefined functionalities (such as path finding, collision detection, and rendering), which would take a lot of time to develop and debug. These functionalities are often available in the shape of menu objects in the development environment. The goal of tools, in general, is to allow developers to quickly prototype and deploy games, while relieving them from common tasks in game development. Typical tools, such as GameMaker, Corona, Unity3D, and RPGmaker, provide an easy-to-use interface and shortcuts for dealing with entity behavior. As long as the developer limits himself to using the components provided, the tools produce

high-performance game code, because of their specific application in the domain of games. For behaviors that are not expressed natively by the tool components, tools often offer scripting languages that allow developers to define custom behaviors. Unfortunately, the expressiveness of such scripting languages is affected by the mechanics of the tools to which they are adapted. Thus, in order to effectively use these languages developers are supposed to make a considerable effort with understanding the mechanics of the chosen tool and to adapt their solutions accordingly. Moreover, the scripting languages used by tools are usually interpreted (like LUA and JavaScript [20]), which considerably affects performance.

General-purpose **languages** (GPLs) are languages that provide powerful, composable, abstractions for expressing general data structures and algorithms. These properties allow such GPLs to express solutions for different kinds of applications including the one of game development. C#, Python, and Objective-C are typical examples of languages used for game development. A typical limitation in using GPLs is in expressing performance patterns [21]. Performance in games is very important, since it is strongly related to the perceived quality of it. Due to this lack of support by GPLs, a game featuring complex data structures and algorithms will require developers to implement optimization by hand, thus increasing the costs of implementing them. Unless the developers have access to large financial resources, the use of a GPL for game development is not a good choice. However, developers can use domain-specific languages (DSLs) for implementing their games, since they are capable not only of expressing domain-specific abstractions easily, but also to perform domain-specific optimization. For instance: the Conceptual Domain Modeling Language (CDM) [22] supports an efficient parallel implementation of a producer/consumer pattern; Inform [23], and Zillions of Games [24] provide domain-specific abstractions focused on the definition of specific genres, such as storytelling or board games; monadic frameworks have been developed to tackle common problems of game development such as combining and defining behaviors that depend on the flow of time and their efficient implementation, to integrate within game engines [25] and reduce code complexity [26]. The only issues of such languages is that in many cases they are prototypes and supported by small communities, which translates into the fact that code that is supposed to work actually does not always because of compiler issues. Moreover, every DSL has its own learning curve, since similarities between DSLs are few (unlike for GPLs): every DSL comes with its own philosophy, constructs, primitives, etc. The advantage

of Casanova 2 over other DSLs is that the latter are usually limited to one or few game genres sharing similar characteristics, while the former can be used to develop any kind of game genre. This will require, of course, to learn Casanova 2 but not other languages because of language limitations.

3.2. Running example

To illustrate the discussions hereafter, we now present a game that contains typical elements that are often encountered in game development. The game consists of a set of planets linked together by routes. A player can move fleets from his planets to attack and conquer enemy planets. Fleets reach other planets by using the provided routes. Whenever a fleet gets close enough to an enemy planet, it starts fighting the defending fleets orbiting the planet. The game can be considered the basis for a typical *Planet Wars* strategy game (such as Galcon [27]). We define a *frame* to be a single update cycle of all the game's data structures.

In our running example, we assume that a `Route` is represented by a data structure containing (*i*) the start and end points as references to `Planets`, and (*ii*) a list of `Fleets` travelling via such route. `Planet` is a data structure containing (*i*) a list of defending `Fleets`, (*ii*) a list of attacking `Fleets`, and (*iii*) an `Owner`. Each fleet has an owner as well. Each data structure contains a method called `Update`, which updates the state of its associated object at every frame. Furthermore, we assume that all the game objects have direct access to the global game state, which contains the list of all routes in the game scenario.

Following the definition of encapsulation given in Section 1, each entity is responsible for updating its own fields in such a way that it maintains its own invariant.

3.3. Design techniques and operations

In our running example the modules are the `Planet` and `Route` classes defined above; *data* are their fields. To support *encapsulation*, in the following implementation each entity is responsible for updating its fields with respect to the world dynamics. The *operations* for each entity are the following:

> **Planet**: Takes the enemy fleets travelling along its incoming routes, which are close to the planet, and moves them into the attacking fleets list. We assume that `Planet` contains a method to check whether a fleet is contained in the list of attacking fleets, called `IsAttackedBy`;

Route: Removes the travelling fleets, which have been placed in the attacking fleets of the destination planet from the list of travelling fleets.

```
class Route
  Planet Start, Planet End,
  List<Fleet> TravellingFleets,
  Player Owner
  void Update()
    foreach fleet in TravellingFleets
      if End.IsAttackedBy(fleet)
        this.TravellingFleets.Remove(fleet)
class Planet
  List<Fleet> DefendingFleets,
  List<Fleet> AttackingFleets
  void Update()
    foreach route in GetState().Routes
      if route.End = this then
        foreach fleet in route.TravellingFleets
          if distance(fleet.Position, this.Position) <
             min_dist &&
             fleet.Owner != this.Owner then
            this.AttackingFleets.Add(fleet)
```

An alternative design, which does not use encapsulation, allows the route to move the fleets close to the destination planet directly into the attacking fleets by writing into the planet fields. In this scenario the route is modifying data related to the planet and the route is writing into a reference to a planet.

```
class Route
  Planet Start, Planet End,
  List<Fleet> TravellingFleets
  void Update()
    foreach fleet in this.TravellingFleets
      if distance(fleet.Position, this.Position) < min_dist
          &&
         fleet.Owner != End.Owner then
          this.TravellingFleets.Remove(fleet)
          End.AttackingFleets.Add(fleet)
```

3.4. Discussion

In our running example a programmer is left with the choice of (i) either using the paradigm of encapsulation, which improves the understandability of programs and eases their modification [28], or (ii) breaking encapsulation by writing directly into the planet fields from an external class, which, as we will show below, is more efficient but potentially dangerous [29].

As far as *performance* is concerned, in the encapsulated version, the planet queries the game state to obtain all routes where endpoints are the planet itself, and for every route selects the enemy travelling fleets that are close enough to the planet. At the same time, a `Route` checks the list of attacking fleets of its endpoints and removes the fleets that are contained in both lists from the travelling fleets. If we consider a scenario containing m planets, n routes, and at most k travelling fleets per route, each planet should check the distance condition for $O(nk)$ ships; thus the overall complexity is $O(mnk)$. The non-encapsulated version checks, for each route, the distance for a maximum of k ships and then directly moves those close to the planet, for which the overall complexity is $O(nk)$. Therefore, the performance on the non-encapsulated version is better. One could argue that adding a spatial index, such as a KD-Tree for fast entities lookup [30], in the planet containing the incoming routes could lead to higher performance; however this would break the *SOLID* (single responsibility, open-closed, Liskov substitution, interface segregation and dependency inversion) principle of Design Patterns, as a planet would contain information on the topology of part of the map. In particular the *Single Responsibility* is violated, as the task of the planet is less deducible.

As far as *maintainability* is concerned, in a game containing planets, many entities might need to interact with each planet (such as fleets, upgrades, and special weapons). Assume that a special action freezes all the activities of a planet. We have to propagate this behavior into the code of all the entities in the game that may interact with a planet, disabling such interactions when the planet is frozen. In the encapsulated version of the code, such behavior needs only be implemented in one place, namely in the planet. In the non-encapsulated version, it must be implemented in each and every entity that may interact with a planet. Moreover, if the developer forgets to make this change even in just one of the entities, the game no longer functions correctly; i.e., bugs associated with planets might actually find their cause in other entities. It is clear that the maintainability of the encapsulated version

of the code is much better than the maintainability of the non-encapsulated version.

The main advantage of using encapsulation is related to the maintainability of code, because encapsulated operations that alter the state of an entity are strictly defined within the entity definition. This helps to reduce the amount of code to maintain in case the entity changes the *normal* behavior of an entity. In our scenario all the activities that alter the planet are inside the planet, so if we remove (or disable) a planet then all its operations are suspended.

What we desire to achieve is the maintainability of encapsulated game code, combined with the performance of non-encapsulated code. In the following sections, we show how this can be achieved with Casanova.

4. Optimizing encapsulation

In this section we introduce the idea of a code transformation technique that changes encapsulated programs into semantically equivalent but more efficient implementations.

4.1. Optimizing lookup

In our running example, the main drawback of the encapsulated version is that each planet has to check all the fleets to see if they are close enough to move into the list of attacking fleets. An optimization can be achieved by maintaining an index `FleetIndex` in `Planet`, containing a list of those `Fleets` that satisfy the attacking property, i.e., being owned by a different player and close enough to the planet. When an enemy `Fleet` is close enough to a `Planet`, it is moved into `FleetIndex` by the `Route`, which stores a list of travelling fleets. When `FleetIndex` changes, it notifies `Planet`, so that `Planet` can update `AttackingFleets`.

A predicate is a conditional statement based on one or more fields of an object of a class A. We can generalize the aforementioned situation by saying that encapsulation suffers from loss of performance whenever an object B needs to update one of its fields depending on a predicate. B stores an index I_A that is used to keep track of all possible objects of class A satisfying the predicate. Any object of A has a reference to B and is tasked with updating the index I_A of B. B checks I_A every time it needs to interact with the instances of A satisfying the predicate.

4.2. Optimizing temporal/local predicates

If we take into consideration the fact that predicates in a simulation are defined on entities (potentially hundreds or thousands) that exhibit similar behaviours (ships, bullets, asteroids, etc.) [31], we can expect that some predicates will exhibit some sort of *temporal locality* on their values. We can group those predicates, and their respective block of code, and apply an optimization that (*i*) keeps their code block inactive in a *fast wake-up* (we use the term *fast* because the implementation uses a dictionary) collection, and (*ii*) activates only those blocks of which the predicate has changed. In general, this would yield a higher performance without asking developers to write the optimization code themselves.

4.3. Casanova 2-level integration

The process described above can be automated at the compiler level as code transformation, since the index creation and management always follows the same pattern, and thus the compiler itself can create and update the required data structures. Casanova is a *Domain Specific Language* oriented towards game development. A program in Casanova is a set of entities organized in a tree hierarchy, of which the root is marked as *world*. Each entity contains a set of fields, a set of rules, and a constructor. An extensive description of the formal grammar and semantics of Casanova can be found in [19]. Casanova 2 (which we use) is a recent iteration of the original Casanova, which does not introduce changes to syntax or semantics. The language ensures that variables are only changed through specific statements; this makes it possible for the Casanova 2 compiler to identify patterns in code that are suitable for optimization. The Casanova 2 compiler applies transformations to the code that preserve the program semantics and optimize the encapsulated implementation by creating and maintaining the required indices. This way the code written by the programmer will gain the benefits of readability and maintainability that encapsulated code brings, without suffering from loss of performance or the necessity to break encapsulation to manage the optimization of data structures. In the next section we present the compiler architecture and the transformation rules.

The Casanova 2 compiler is written in F# and offers a modular extensible architecture made of a series of distinct layers, each performing a transformation task. Transformations initially add new information by means of analysis and inference. After analysis the compiler starts removing infor-

mation in preparation for the code-generation step. This is the synthetic[2] part of the compiler. To let the compiler support our technique, we need to implement and add a new layer whose task is to provide the compiler with information regarding the dependencies (necessary to implement the idea above), which is then used during the code-generation step.

5. Implementation details

In this section we show how to select the predicates and the associated blocks of code that can be optimized.

Most games represent simulations of some sort. A property of simulations is a certain *temporal locality* of behaviors [31]. This translates to the fact that some predicates tend to have a high chance of no value change between frames. To reduce the amount of interactions and achieve better performance, we could optimize those predicates that exhibit temporal locality. This can be performed by either using code analysis based on statistics on the average inactive time or by manually annotating the parts that stay inactive for a long time (we discuss this also in Section 9).

We will refer to a predicate on fields that do not change at every frame as *Interesting Conditions* (ICs). These predicates are stored in a data structure called the *Interesting Condition Data Structure* (ICDS).

Dealing with ICs adds an additional layer of complexity to the game. The execution of game mechanics tends to be very frequent (we may expect that some mechanics will be executed potentially hundreds of times per second), so interacting frequently with ICs affects the game performance due to the complexity of the data structure.

ICs are used to identify which blocks of code can be suspended and resumed with little overhead. We use ICs at compile time to generate code that is able (through the support of a specific data structure) to suspend and wake up with little overhead. This is schematically shown in Figure 1.

5.1. Casanova overview

In Casanova, the state of a game changes only upon the execution of a *rule*. A rule is a block of code acting on a subset of the entity fields called

[2]In this context we are using the term "synthetic" in opposition to "analytic" to underline that the compiler has two phases: one that analyses the source code, and one that produces the target code by exploiting the information gathered by the analysis phase.

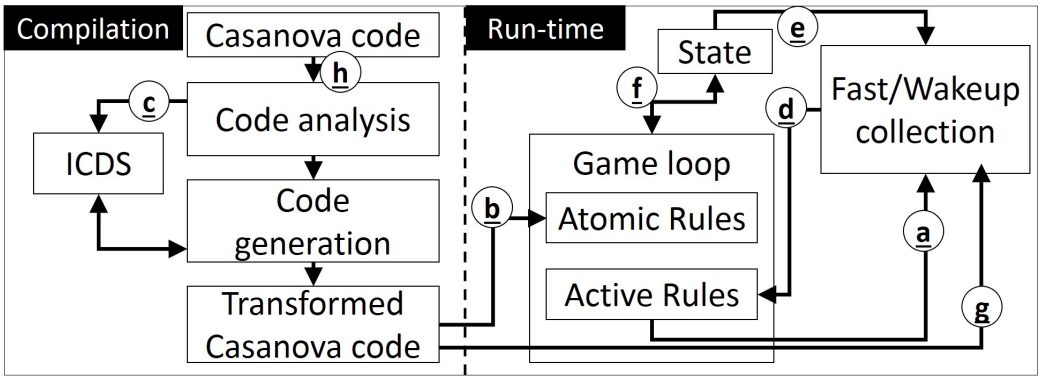


Figure 1: System Configuration

domain, which has at least one `yield` statement and zero or more `wait` statements. The former updates the value of the fields of an entity; the latter suspends the evaluation of the rule until its condition is met, temporally affecting the fields update. The rule body is re-executed once the end is reached.

An example of a rule that illustrates the `wait` statement (which specifies that a shield is repaired when it gets damaged) is the following :

```
rule Shields =
  wait Shields < 0
  ...
  yield Shields + 1
```

5.2. Compilation - Recognizing ICs in Casanova

From here on we will refer to the `wait` predicate as an IC, since its value affects the update of an entity with respect to the flow of time.

We also include query conditions in our IC taxonomy. We can think of a query as an entity containing a list of valid query elements that satisfy the `where` condition. An element adds itself to the valid query elements only if it satisfies the query `where` condition (this is done by adding to its rules a rule that starts with a `wait` on the query condition and ends with a `yield` that appends itself to the valid query elements).

An example of a rule with a query (which selects ships that are not destroyed) is the following:

```
rule Ships = yield [from s in Ships do
```

```
                    where s.Life > 0
                    select s]
```

The effect of a `yield` is to suspend the execution of the rule for one frame and to assign the selected query elements to the selected field. To achieve the optimization as described in the previous section, the compiler uses an optimization analyzer (composed by a code analyzer and a code generator as shown in Figure 1(h)), which requires the identification of ICs in the code. This is discussed next.

Casanova allows interaction with external libraries and frameworks such as the .NET framework. Because the analyzer cannot infer the temporal behavior of external libraries, we add the restriction that an IC must be fully dependent on Casanova data types. The restriction is necessary because the analysis will lead to alterations in the structure of the game code and field creation, update, and access. Given the informal considerations above, we introduce the following definitions:

- A *suspendable statement* is either a `wait` or a `yield`;
- A *suspendable rule* is a rule containing a suspendable statement. A suspendable rule is *interesting* (ISR) if the `wait` argument is an IC or a `yield` on a query.
- An *atomic rule* is a rule that does not contain suspendable statements.

We now present two algorithms that respectively check if a predicate is affected by an atomic rule (Algorithm 1) and build the ICDS (Algorithm 2)[3]. For brevity we do not present the procedure to check if a rule is an ISR, which can be done by simply looking at the syntax tree of the rule body.

Given a Casanova program, we build the ICDS data structure as follows: we iterate over every entity; for every rule in each entity, if the rule is suspendable, interesting and the predicate does not contain fields that are affected by an atomic rule, we add the entity, the rule index, the rule domain, and the predicate to the ICDS (See Figure 1(c)).

We now focus on the identification of interesting conditions that exhibit temporal locality.

[3]In the algorithm `r.index` refers to an incremental index assigned by the compiler to each rule for each entity.

Algorithm 1 Check if a predicate is affected by an atomic rule

```
function ATOMIC(p)
    E is the set of entities.
    DFA ← ∅
    for e ∈ E do
        R is the set of rules in e
        for r ∈ R do
            if r is an atomic rule then
                for f ∈ r.domain do
                    DFA ← DFA ∪ {(e, f)}
                end for
            end if
        end for
    end for
    D ← set of (entity, field) in the predicate p.
    return ∃x ∈ D : x ∈ DFA.
end function
```

Algorithm 2 ICDS construction

```
function BUILDICDS( )
    ICDS ← ∅
    E is the set of entities.
    for e ∈ E do
        R is the set of rules in e
        for r ∈ R do
            if r is an ISR then
                p is the first interesting condition of r
                if not ATOMIC(p) then
                    ICDS ← ICDS ∪ {(e, r.index, r.domain, p)}
                end if
            end if
        end for
    end for
    return ICDS
end function
```

5.3. Run-time efficient sleep/wake-up system

We use the data structure generated by the analyzer to produce two distinct kinds of rules: atomic rules (see Figure 1(b)) that are run every frame, and suspendable rules (see Figure 1(g)). Every suspendable rule depends on an IC. Because of the assumed property of temporal locality of rules that contain ICs, they do not need to run at every frame. Therefore the game program should activate and deactivate rules as needed at run time. The game needs to: (*i*) activate a suspendable rule when its IC changes value, and (*ii*) deactivate a suspendable rule when its IC is not satisfied (i.e., when it is `false`). The game keeps a rule active as long as the evaluation of its IC is `true`. Suspendable rules differ from classic atomic rules in Casanova since suspendable rules may become inactive, i.e., they do not run during every update in the game loop.

We define the *Object Set* (`OBS`) as the set of pairs made of an instance of an entity and its field, that appear as arguments in an IC. Information used to build an `OBS` is collected by using the ICDS. The idea behind the optimization is that, whenever the field of an element of `OBS` changes during the game loop (see Figure 1(f)), we activate the corresponding *Interesting Suspendable Rule* (ISR) `R` by triggering it (see Figure 1(e)).

We implement the previous behavior by means of dictionaries that keep track of the dependencies among `OBS` and `R`. We use dictionaries in this implementation since they exhibit the best asymptotic complexity with respect to the following operations: check, add, remove, and iterate. From now on we will refer these dictionaries as *Dictionary of Entity-Predicate* `DEP`.

We use the static information from the ICDS (see Figure 1(c)) to refer to the appropriate dictionary, based on the structure of the IC, to generate unique names for dictionaries. For every field in the predicate, we combine the name of the type of the object containing the field, the name of the field itself, the entity containing the ISR, and the ISR index.

As key we use a pair made of the reference to the object containing the field of the IC and the field itself. As value we store a collection of pairs made of the instance of the entity containing the ISR and the ISR index. We use a collection because it might be the case that one or more instances of the same entity type are pending on the same specific object field. In the example below, the rule in `E` waits on a field `X` in the `world`, and the `world` contains a collection of instances of `E`. When `X` changes (the statement `wait 10` means wait for 10 seconds), all the rules of each instance of `E` waiting for `X` must be resumed.

```
world W =
  X : int
  L : List<E>
  rule X =
    wait 10
    yield X + 1
          ...
entity E =
  ...
  rule Y =
    wait world.X % 2 = 0
    ...
```

An entry of the dictionary in the example would be `(world,X), {(L[0],rule Y)}`.

5.4. Suspendable rules instantiate, destroy, and update

In order to maintain the suspendable rules, we identify three stages that represent the life cycle of a suspendable rule:

- **On creation**: when we instantiate an element of which a field appears in one of the pairs of `OBS`, we use the instance and the field itself as a key to populate all its DEPs with an empty collection as value. When we instantiate an entity, the rules of which are targeted by an IC, we add the pair made of the entity instance and each targeted rule as a value in its DEPs;
- **On destroy**: when an instance appears either as a value or a key in one of DEPs, we remove all the occurrences of the instance in DEPs;
- **On update**: when a field of an IC changes, we notify the entities pending on it. After generating the IC data structure, we can safely refer to the dictionaries relying on the fact that the generated code is sound and will not produce errors at run time. As a consequence of a notification, the ISRs involved in the notification will be activated during the next frame (if they were inactive). We add them to a collection representing the active rules of the entity containing the involved ISRs (see Figure 1(d)). We group instances of the same target type into the same collection to achieve better performance (we iterate the active rules all at

the same time per type instead of iterating them while iterating each entity). We store a collection in the `world` entity that contains all the suspended rules that are currently active. These rules are grouped by their respective entities.

Rules in Casanova are translated at compile time into a series of switches without nesting within functions that return `void`. ISRs return `Done` when the evaluation of their IC is `false` (stay inactive) or `Working` when the evaluation of their IC is `true` (go active) or we are still busy with the execution of the block after the IC. When a suspendable rule gets suspended, i.e., its evaluation returns `Done`, we simply remove it from the active rules collection (see Figure 1(a)).

Notice the similarity between Casanova and the Aspect-Oriented Programming paradigm: interruptions and yields share the same philosophy of cross-cutting concerns and aspects. When using a `wait`, a developer simply declares the concern of suspending the code with respect to some condition, and the compiler acts as a weaver inlining the code for the optimization (our aspect) discussed above. When using a `yield`, a developer declares the concern of updating an attribute of an entity, and the compiler realises the wake up system for all the wait statements depending on that attribute.

5.5. Query interpretation

We transform a query into semantically equivalent code where every entity appearing in the `from` expression (*source*) adds itself from an index stored in the entity containing the query (*target*). We add a source entity in the target index only if the condition is `true`. An entity removes itself if the condition is `false` and the entity is present in the index. This is done by generating a rule that waits for the condition to be `true` in the target entity. Applying our optimization to queries means that we do not need to iterate conditions every frame: we keep the rule suspended until the condition changes its value.

6. Networking in Casanova 2

In this section we introduce the basic concepts of the implementation of multiplayer game development for Casanova 2. This implementation aims to relieve the programmer of the complexity of hard-coding the network implementation for an online game, while preserving encapsulation in code. We show that code analysis is required to generate the appropriate network

primitives to send and receive data. Finally, we present a simple multiplayer game to show a concrete example.

6.1. Introduction

Adding multi-player support to games is a highly desirable feature. By letting players interact with each other, new forms of gameplay, cooperation, and competition emerge without requiring any additional design of game mechanics [32]. This allows a game to remain fresh and playable, even after the single player content has been exhausted. For example, consider any modern AAA (AAA refers to games with the highest development budgets[33]) game such as *Halo 4*. After months since its initial release, most players have exhausted the single player, narrative-driven campaign. Nevertheless the game remains heavily in use thanks to multiplayer modes, which in effect extended the life of the game significantly. This phenomenon is even more evident in games such as *World of Warcraft* or *EVE*, where multiplayer is the only modality of play and there is no single-player experience.

Challenges. Multi-player support in games is a very expensive piece of software to build. Multiplayer games are under strong pressure to have very good *performance* [34]. Performance is both expressed in terms of CPU time and in bandwidth used. Also, games need to be very *robust* with respect to transmission delays, packets lost, or even clients disconnected. To make matters worse, players often behave erratically. It is widespread practice among players to leave a competitive game as soon as their defeat is apparent (a phenomenon so common to even have its own name: "rage quitting" [35]), or to try to abuse the game and its technical flaws to gain advantages or to disrupt the experience of others.

Networking code reuse is quite low across titles and projects. This comes from the fact that the requirements of every game vary significantly: from turn-based games that only need to synchronize the game world every few seconds, and where latency is not a big issue, to first-person-shooter games where prediction mechanisms are needed to ensure the smooth movement of synchronized entities, to real-time strategy games where thousands of units on the screen all need to be synchronized across game instances [36]. In short, previous effort is substantially inaccessible for new titles.

Encapsulation suffers from this ad-hoc nature of the implementation of the networking layer in multiplayer games. Indeed managing the information about game updates over a network requires each game entity to interface

the game logic code with network connection and socket objects, data transmission method calls such as send and receive, and support data structures to manage traffic and track the status of common protocols. This happens because each game entity must provide the following functionality in order to work in a multiplayer game:

- Update the logic in the fashion of a singleplayer counterpart.
- Choose what data is necessary to send over the network and create the message containing this information.
- Choose what data can be lost and what data must always be received by the other clients.
- Periodically check if incoming messages contain information that needs to be read and to perform specific updates.

Combining these requirements together within the same entity breaks encapsulation because now the logic of the entity and lots of spurious details only relevant to the networking implementation are mixed together, resulting in a highly noisy program. Maintenance then becomes very hard, as every change in the game logic must also be reflected in the networking implementation.

Existing approaches. Networking in games is usually built with either very low-level or very high-level mechanisms. Very low-level mechanisms are based on manually sending streams of bytes and serializing only the essential bits of the game world, usually incrementally, on unreliable channels (UDP). This coding process is highly expensive because building by hand such a low-level protocol is difficult to get right, and debugging subtle protocol mismatches, transmission errors, etc. will take lots of development resources. Low-level mechanisms must also be very robust, making the task even harder.

High-level protocols such as RDP, reflection-based serialization, frameworks (such as Pastry, netty.io), etc. can also be used. These methods greatly simplify networking code, but are rarely used in complex games and scenarios. The requirements of performance mean that many high-level protocols or mechanisms are insufficient, either because they are too slow computationally (especially when they rely on reflection or events) or because they transmit too much data across the network.

6.2. Motivation

To avoid the problems of both existing approaches, we propose a middle ground. We observe that networking fundamental abstractions upon which the actual code and protocols are built do not vary substantially between games, even though the code that needs to be written to implement them does. The similarity comes from the fact that the ways to serialize, synchronize, and predict the behaviour of entities are relatively standard and described according to a limited series of general ideas. The difference, on the other hand, comes from the fact that low-level protocols need to be adapted to the specific structure of the game world and the data structures that make it up. Until now, common primitives have not been syntactically and semantically captured inside existing domain-specific languages for game development [37]. Using the right level of abstraction, these general patterns of networking can be captured, while leaving full customization power in the hand of the developer (to apply such primitives to any kind of game).

6.3. Related work

In the following we discuss some existing networking tools used in game development and we highlight some issues that arise from their use.

The Real time framework (RTF). RTF [38] is a middleware built for C++ to relieve the programmer from dealing with data compression. It is more flexible than solutions based on game engines or hand-made implementations, since it automates the process of data transmission. Moreover, it supports distributed server management. Unfortunately, this solution has several flaws:

- All entities must inherit from the class `Local` and the semantics of the position is pre-determined, often clashing with rendering or physics.
- Platform independence requires that the programmer uses RTF primitive types.
- Data transmission automation requires that all game entities inherit the class `Serializable`.
- Being a middleware, RTF is not aware of what games are going to use it for (every game comes with different data structures). Thus, the developer is tasked to include in his code also logic to update the RTF layer, in order to keep the game updated over the network.

Network scripting language (NSL). NSL [39] provides a language extension based on a send-receive mechanism. Moreover it provides a built-in client side prediction (a feature missing in existing highly concurrent and distributed languages such as Stackless Python [40] and Erlang [41]), which is periodically corrected by the server.

Unreal Engine/Unity Engine. Unreal Engine [42] and Unity Engine [43] are commercial game engines supporting networking. Both Unity and Unreal Engine use a client-server approach. In Unreal Engine, the server contains the "true" game state, and the clients contain a "dirty" copy, which is validated periodically. It is possible to define entities (actors in Unreal Engine jargon) that are replicated on the clients. Whenever a replicated actor changes on the server, this change is also reflected on the clients. Additional customization can be achieved through Remote procedure calls (RPCs) of three kinds.

- The function is called on the server and executed on the client. This is used for game elements that do not affect gameplay, such as creating a particle effect when a weapon is fired.
- The function is called on the client and executed on the server. This is useful for events that affect the other clients and should be validated by the server.
- The function is executed in multi-cast, meaning that the server calls the function and that it is executed on both the server and all the clients.

The Unity Engine uses a similar approach based on networking components, synchronized at every frame, and RPC's to define custom synchronization events.

Unfortunately, customization comes at the cost of the level of detail that developers must face. Using RPC's require a deep knowledge of the engine and writing lots of code, as discussed in Section 3.1.

7. Networking architecture

In this section we introduce a small example that addresses the requirements of designing a multiplayer game. We then present an architecture that aims to fulfil these requirements.

7.1. The master/slave network architecture

We chose to implement the networking layer in Casanova 2 by using a peer-to-peer architecture for the following reasons:

- Server-client architectures are more reliable but suitable only for specific genres of games (mostly Shooter games), while other genres, such as Real-time strategy games or Online Role Playing Games, use P2P architectures.
- We do not have to write a separate logic for an authoritative game server, which has to validate the actions of clients.

Casanova will provide a generic tracking server, which is run separately from the main program. The tracking server is a thin service that connects players participating in a single game, and helps with forwarding the network traffic through NATs (Network Address Translation).

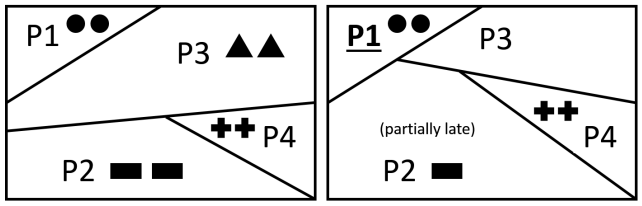
Each client maintains a local copy of the `world` entity and has direct control over a single portion of it. Instances belonging to such as portion are seen as *master* by this player, who is always allowed to directly change the state of the master instances without having to validate this state change by synchronizing with other players through the network.

Each client also maintains a portion of the world that is not directly under his control. Instances belonging to such as portion are seen as *slave* by this player, who is only allowed to *predict* the local state of the instances and, whenever he receives an update from their masters, must correct this prediction according to the data contained in the received messages. The slave part of the world is thus maintained passively by the client: the only active part is predicting the evolution of the entity state and correcting it whenever he receives an update by its master.

For this purpose, we extend the syntax of Casanova rules by allowing them to be marked with the modifiers `master` and `slave`. These rules are executed respectively on master and slave entities. Note that it is still possible not to mark a rule with these modifiers, which means that the rule is always executed independently of the fact that the entity is either master or slave on that particular client. We also allow to mark a rule as `connecting` and `connected`. These rules are triggered only once respectively when a new client connects and when the clients detect a new connection.

Casanova also provides primitives to send (reliably or unreliably) and receive data. A schematic representation of this architecture can be seen in Figure 3.

Figure 2: Representation of the game world in a networking scenario



(a) Unknown correct game state when P3 joins the game.

(b) Networking game state seen from the point of view of P1. P2 is partially synchronized, P4 is fully synchronized, and P3 is a new client that is late and is still sending its data

Note the aim of this architecture is to provide language-level primitives to describe the networking logic. This means that the compiler will be able to generate code compatible with the low-level network libraries that provide transmission functions over the network channel without having to change Casanova code in the program. In our implementation, we chose the .NET library `Lidgren`, which is widely used also in commercial game engines such as Unity3D and MonoGame, but nothing prevents the compiler to be expanded in order to target other similar libraries for other languages, such as jgroups [44].

7.2. Case study

Let us consider a simple shooter game where each player controls a space ship. Players can move forward, backward, and rotate the ship to change direction. Moreover, they can use the ship lasers to shoot other players. If a laser hits an enemy ship, we increase the player's score. Designing such a game requires to address the following issues, depicted by the schematic representation in Figure 2:

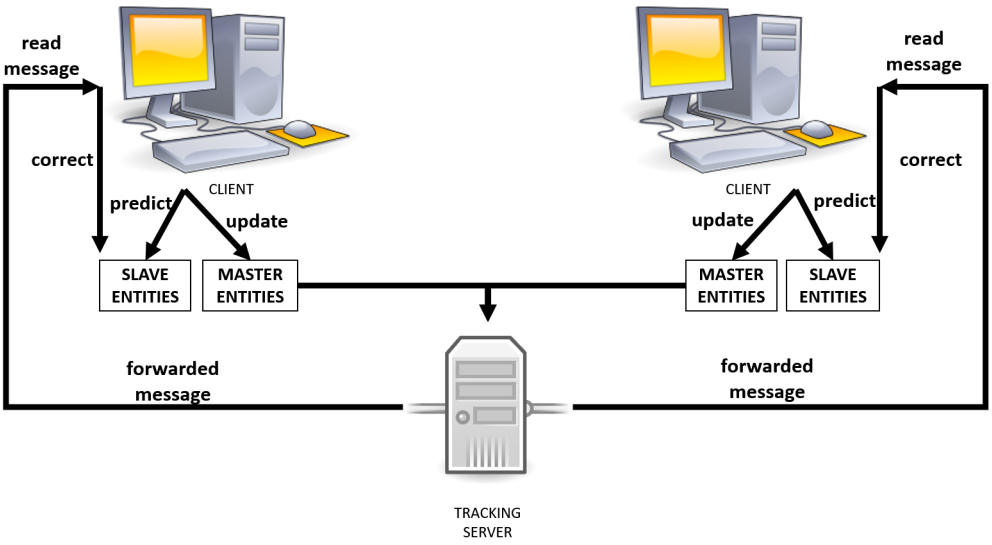


Figure 3: master/slave architecture

1. Each player must maintain a local version of the game state (world). In order to avoid to flood the network with messages, all the copies are not fully synchronized at each frame, thus they are slightly different and each client knows the latest version of only part of the copy.
2. A player `connecting` to an existing game must be able to receive the latest update of the game state and send the new ship he will control to existing players in the game.
3. A player already `connected` to the game must detect a new connection and send his master portion of the game state.
4. Each player must be able to control only one ship at a time. This means that the part of the game logic that processes the input and modifies the spatial data of the ship (position and rotation) should only be executed on the ship controlled by the player and not on the local copies of other players' ships. This means that each player sees as `master` only one ship instance.
5. Each player must send the updated state of the ship he controls to the other players after executing the local update. To achieve better performance over the network, the data is not sent at every update, but with a lower frequency.

6. Each player must receive the updated state of `slave` ships controlled by other players. In this phase, we must take into account that, as explained above, not every update is sent, so the player should "predict" what will happen during the game frames in which he does not receive an update.

7.3. Implementation

Each of the scenarios described above requires specific language extensions. These extensions identify connection, ownership (master/slave), and various send and receive primitives. In this section, we introduce each primitive by using a multiplayer game example [4]. We now give an implementation of the shooter game presented above, using the extended version of Casanova 2 with network primitives.

The `world` contains a list of ships controlled by each player.

```
world Shooter = {
  Ships  : [Ship]
  ...
}
```

Each `Ship` contains a position, a rotation, a collection of shot projectiles, and the score.

```
entity Ship = {
  Position    : Vector2
  Rotation    : float32
  Projectiles : [Projectile]
  Score       : int
  ...
}
```

Each `Projectile` contains its position and velocity.

```
entity Projectile = {
  Position : Vector2
  Velocity : Vector2
  ...
```

[4] The game source code and executable can be found at `https://github.com/vs-team/casanova-mk2/wiki/Networking-extension`

```
}
```

7.3.1. Connection

When a player connects, we must consider two different situations: (*i*) a player is already in the game and must send the current game state to the connecting players, and (*ii*) the player who is connecting needs to send the ship he will instantiate and control (its initial state). Both the players in the game and the connecting one must receive the game states that are sent. For this purpose we introduce two additional modifiers, `connecting` and `connected`, that can be added to rule declarations to mark their role in the multiplayer logic.

Connecting. A rule marked with `connecting` is executed once when a player joins the game for the first time. In our example, the player should send his initial state (the created ship) to the other players. We use the primitive `send_reliable` because we must be sure that eventually all players will be notified of the ship creation.

```
world Shooter = {
  ...
  rule connecting Ships =
    yield send_reliable Ships
}
```

Connected. A rule marked with `connected` is run whenever a new player joins the game by all existing players. When this occurs, each player sends its ship. The system will take care to send only the ship controlled locally by the player itself for each player. The rule will use the `send_reliable` primitive for the same reason explained in the previous point.

```
world Shooter = {
  ...
  rule connected Ships =
    yield send_reliable Ships
}
```

Note that even if the code is the same, the semantics of the two rules are different. The first one is executed by the player joining the game, who locally instantiates its `Ship` and must send its list of `Ships` (containing only

the local instance) to the other players. The second one is executed by all existing players who must share with the joining player the list of existing ships.

7.3.2. Master updates

As explained above, each client manages a series of local game objects (called *master objects*) that are under its direct control. The other clients read passively any update done on those instances and update their remote copy (*slave objects*) accordingly. We mark rules affecting the behaviour of master objects as `master`. In our example, the following situations are run as master: (*i*) synchronizing the ships among players, (*ii*) updating the ship and projectiles spatial data, and (*iii*) creating and destroying projectiles.

1. Each player is tasked to maintain the list of Ships in the world. This requires to receive the updated list from other players and to store the new value in a master rule. Indeed the world is a special case of an entity that is shared among players, and not directly owned by somebody. Each ship contained in that list and received from other players will be treated appropriately as slaves, while the only one owned by the current player will be under his direct control. In this rule we use `let!`, which is an operator that waits until the argument expression returns a result and then binds it to the variable. The symbol `@` stands for list concatenation. The rule uses `receive_many`, which receives and collects the list of sent ships by the other players.

```
world Shooter = {
  ...
  rule master Ships =
    let! ships = receive_many()
    yield Ships @ ships
}
```

2. The master version of the ship update reads the input of the player and moves (or rotates) the ship if the appropriate key is pressed. Note that this part must be executed only on a master object, because we want to allow each player to control only the ship he owns and instantiates at the beginning of the game. Below we show just the rule to move forward; the other movement and rotation rules are analogous. We use an *unreliable send* because it is acceptable to lose an update of

the position during a certain frame: shortly after, there will be a new update.

```
entity Ship = {
  ...
  rule master Position =
    wait world.Input.IsKeyDown(Keys.W)
    let vp = new Vector2(Math.Cos(Rotation),
                         Math.Sin(Rotation)) * 300.0f
    let p = Position + vp * dt
    yield send p
}
```

We do the same for projectiles, except the projectile position is continuously updated and synchronized over the network without having to wait that a key is pressed.

3. Creating a new projectile happens when the player shoots. A ship keeps track of the projectiles it has shot so far, and adds a new one to the list of the existing projectiles. The updated list is sent to all players with the new instance of the projectile (which is added as a new head of the list with the operator `::`). Here it is better to precise the semantics of the `yield` in conjunction with the use of networking primitives. A `yield` requires that the written value is type-compatible with the domain of the rule. Thus, when used with a `send` primitive, we must pass as argument a list. The system will ensure, for performance reasons, that the generated code only sends the new items added to the list. This semantics is defined as such for two main reasons: (*i*) when sending the new projectiles we must also update the list in local (and given the immutability of Casanova we must replace the existing one), and (*ii*) because in this way the programmer can focus on the logic of the game as if it were a single-player game without worrying of network-specific details. Note that the last `wait` forces the player to release the key before shooting again (semi-automatic fire). Removing that check would spawn multiple projectiles consecutively, which is not a wanted behaviour.

```
entity Ship = {
  ...
  rule master Projectiles =
    wait world.Input.IsKeyDown(Keys.Space)
```

```
    let vp = new Vector2(Math.Cos(Rotation),
                         Math.Sin(Rotation)) * 500.0f
    let projs = new Projectile(Position, vp) ::
       Projectiles
    yield send_reliable projs
    wait not world.Input.IsKeyDown(Keys.Space)
}
```

Filtering the colliding projectiles and updating the score is run as a master rule. The rule computes the set difference between the ship projectiles and the colliding projectiles and updates the list of projectiles, sending them through the network as well. Even in this case, the network layer sends only the information about the projectiles to remove. Note that the score is managed by each player locally, as it does not require to be synchronized (we do not print the other players' scores. Doing so would indeed require to also send the score).

```
entity Ship = {
  ...
  rule master Projectiles, Score =
    let collidingProjs =
      [for p in Projectiles do
       let ships =
         [for s in Ships do
          where
           s <> this and
           Vector2.Distance(p.Position,s.Position) <
              100.0f
          select s]
       where ships.Count > 0
       select p]
    let newProjectiles = Projectiles - collidingProjs
    yield send_reliable newProjectiles,
          Score + collidingProjs.Count
}
```

7.3.3. Managing remote instances

The game objects that were not instantiated by a client, but received from another client, are *slave objects* and must be synchronized differently than

master objects. For this purpose, a rule can be marked as `slave`. In our example, we use slave rules in the following situations: (*i*) synchronizing other players' ships and projectiles spatial data, and (*ii*) projectiles instantiated by other players.

1. Every remote projectile and ship is synchronized locally by a rule, which tries to `receive` a message containing updated spatial data. Below we provide the code to update the position of the ship; the synchronization of other spatial data is analogous.

```
  entity Ship = {
  ...
  rule slave Position = yield receive()
}
```

2. When a projectile is instantiated remotely, we have to receive it and add it to the list of projectiles. We use `receive_many` because the new projectiles are added to a list. This case also supports the situation where a ship could shoot multiple projectiles at the same time.

```
entity Ship = {
  ...
  rule slave Projectiles =
    let! projs = receive_many()
    yield projs @ Projectiles
}
```

In this scenario is important to discuss the atomicity of these transmissions: in the context of network games, reliability is often sacrificed for better network performance, so most of the data transmissions are unreliable (like in the case of the ship position). This means that we have no guarantee that the message will be received. Several issues can arise from this situation: for example, if a player fails to receive the position of the ship, then it might miss a collision with a projectile. This is a well-known issue in several shooter games and out-of-sync errors might happen during a multiplayer game. However, ensuring that all the data transmissions are reliable might affect network performance to the point that the game would become unplayable because of the network overload.

Casanova 2 allows the programmer to decide whether the transmission should be reliable or not and experiment with the effect of a reliable trans-

mission versus an unreliable one that does not overload the network. For example, the updated list of projectiles, after a collision, is always sent in a reliable way. This is acceptable because collisions are not so frequent. This is not true for the ship position, since movements are very frequent and mostly happen at every frame, thus it is something that should not be sent reliably at every frame.

Furthermore, we want to focus the attention on the implicit relationship between this networking architecture and the encapsulation: as shown for instance in the examples where the ship shoots a projectile, we ensure encapsulation by keeping a semantics that filters completely the details about networking. The programmer only worries about the logic of adding a new projectile, while the details of the network transmission are hidden. A handmade implementation is usually prone to break this separation of concerns because the transmission logic is tightly coupled within the game logic itself.

8. Evaluation

In this section we evaluate the performance of our approach. A comparison on the same Casanova game code between the non-optimized implementation, the optimized one, and an implementation in C#, will be shown and discussed in terms of run-time performance and code complexity.

8.1. Experimental setup

In order to get a systematic evaluation of the proposed approach to encapsulation, a generic game is considered, in which a group of entities are spawned every `K` seconds and stay inactive for a random amount of time, between 5 and 10 seconds. Then they are activated and start moving for a randomly determined amount of time, between 4 and 8 seconds. Finally, they are destroyed, by triggering a condition in the entities. For the evaluation, additional conditions are added (with different timers), in order to make the simulation dynamics more articulated and "heavy" in terms of amount of code to run.

In this experiment, we compare the code generated by the Casanova compiler versus our optimization built in the Casanova compiler, and an idiomatic implementation in the C# language (a commonly-used language for building games). We also ran the games with two different front ends, namely Unity3D and MonoGame, both using .NET. For each test we measure the time (in milliseconds) that the game takes to fully complete a game iteration

(i.e., updating all the entities in the game). We did not include the time it takes to render the game screen, since rendering is not affected by our optimization, though it might affect the performance measure.

8.2. Performance evaluation

Table 2 shows the performance results. As we can see, in both cases, the performance of our optimized Casanova 2 code is higher than the one of non-optimized implementation, and the idiomatic C# implementation. Using Unity3D, the optimized code is one order of magnitude faster than the non-optimized code. Using MonoGame, the optimization is faster but on the same order of magnitude. The difference is due to the implementation of the underlying frameworks.

Table 1: Code lines comparison for a singleplayer game

Original language	Generated language	Optimized code	Lines
Casanova	-	-	45
Casanova	C#	No	139
Casanova	C#	Yes	327
C#	-	-	88

8.3. Code size evaluation

Table 1 shows the code length for each implementation. Casanova 2 game code needs about half the lines of code compared to the idiomatic C# implementation for single player games. When comparing networking code, the difference is one order of magnitude (see Table 3). The intermediate code that the Casanova 2 compiler creates (which is C# code) is considerably

Table 2: Running time comparison for a singleplayer game

Platform	Language	Optimized	Performance
Monogame	Casanova	No	0.0159 ms
	Casanova	Yes	0.0098 ms
	C#	-	0.0147 ms
Unity3D	Casanova	No	0.0257 ms
	Casanova	Yes	0.0085 ms
	C#	-	0.1642 ms

Table 3: Code lines comparison for a multiplayer game

Language	Lines
Casanova	126
C#	1257

longer due to the presence of support data structures. With increasing code complexity, we may expect the original Casanova 2 code to remain compact, while the generated code will increase rapidly in size, with additional data structures and associated logic code. Writing such optimized code by hand is a daunting and expensive task.

9. Conclusions

Game developers often have to choose between maintainability of their code and speed of execution, a choice that more often than not favors speed over maintainability. By using encapsulation, game code may be written in a maintainable way, but compilation of encapsulated code in general-purpose languages often leads to slower games. We proposed a solution to the loss of performance in encapsulated programs using automated optimization at compile time. We presented an implementation of this solution in the Casanova 2 language. We showed that our approach transforms encapsulated code, through extensive automated optimization, into a high-performance executable that easily rivals the speed of a C# implementation. Moreover, we showed that Casanova 2 code needs about half the lines of code as the C# implementation, with even more dramatic results if we consider well-known complex and verbose network code: good primitives for networking preserve encapsulation because they do not require polluting the identity of an entity with network specific information that it is not related to the game logic of the entity itself. Of course networking does impact the logic of the entity, but this should be reflected by minimal code adjustments. Our research, which is still in its initial stage, requires more (and more complex) samples and further investigation. Still, preliminary experiments suggest that our approach allows game developers to write clear, readable code, which is both high-performance and maintainable.

In the near future, we will add a dynamic analysis of the game to automatically identify entities that exhibit temporal locality. Such an analysis must be done while playing the game to keep track of the amount of time

that each rule stays inactive, and periodically the game must be recompiled to generate or remove the optimization code for an entity based on the temporal information generated by the analyser.

Furthermore, we are actively developing a meta-compiler that generates predictable, high-performance code that makes very little to no use of dynamic constructs. A meta-compiler is a software that takes as input the definition of a programming language and a program written in that language and outputs executable code. This meta-compiler features a programmable, higher-kinded type system that feeds an aggressive code inliner. Our meta-compiler is designed to be agnostic with respect to the architecture of one's library for generating code: it could be used for Aspect-Oriented Programming, the inlining of monad transformers, query optimizations, and even other forms of static code analysis such as abstract interpretation or model checking. Such work is unfortunately very complex and as such is not yet in a state to be used in practice with the results of the current paper. We plan on rebuilding the existing Casanova compiler, and all of its optimizations, within the meta-compiler itself. A preliminary result can be found in [45].

[1] E. software association (esa), Essential facts about the computer and video game industry, `http://essentialfacts.theesa.com/Essential-Facts-2016.pdf` (2016).

[2] CMP Media Game Developers Conference, `http://www.gdconf.com/` (2004).

[3] M. Prensky, Computer games and learning: Digital game-based learning, Handbook of computer game studies.

[4] A. Stapleton, Serious games: Serious opportunities, in: Australian Game Developers Conference, Academic Summit, Melbourne, 2004.

[5] P. Petridis, I. Dunwell, S. De Freitas, D. Panzoli, An engine selection methodology for high fidelity serious games, in: Games and Virtual Worlds for Serious Applications, IEEE, 2010.

[6] M. Lewis, J. Jacobson, Game Engines in Scientific Research, Communications of the ACM 45 (2002) 27–31.

[7] K. Rocki, M. Burtscher, R. Suda, The future of accelerator programming: Abstraction, performance or can we have both?, Symposium on Applied Computing, ACM, 2014.

[8] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, K. Olukotun, Delite: A compiler architecture for performance-oriented embedded domain-specific languages, Transactions on Embedded Computing Systems.

[9] K. Beck, Extreme programming explained: embrace change, Addison-Wesley Professional, 2000.

[10] E. Collar Jr, R. Valerdi, Role of software readability on software development cost, Tech. rep., Western Connecticut State University (2006).

[11] ISO/IEC/IEEE, ISO/IEC/IEEE 24765 - Systems and software engineering - Vocabulary, Tech. rep., ISO/IEC/IEEE (2010).

[12] M. Claypool, K. Claypool, Perspectives, frame rates and resolutions: it's all in the game, in: International Conference on Foundations of Digital Games, ACM, 2009.

[13] H. Takeuchi, I. Nonaka, The new new product development game, Harvard business review 64 (1) (1986) 137–146.

[14] A. Marx, Interactive Development: The New Hell, Variety 354 (1994) 1.

[15] G. Zhou, Partial evaluation for optimized compilation of actor-oriented models, ProQuest, 2008.

[16] M. Abbadi, F. Di Giacomo, A. Cortesi, P. Spronck, C. Giulia, G. Maggiore, High performance encapsulation in Casanova 2, in: Computer Science and Electronic Engineering Conference (CEEC), 2015 7th, IEEE, 2015, pp. 201–206.

[17] D. Ungar, R. Smith, C. Chambers, U. Hölzle, Object, message, and performance: how they coexist in Self, Computer 25 (10) (1992) 53–64.

[18] J. Richter, CLR via C#, Pearson Education, 2012.

[19] G. Maggiore, Casanova: a language for game development, Ph.D. thesis (2013).

[20] E. Anderson, A classification of scripting systems for entertainment and serious computer games, in: Games and Virtual Worlds for Serious Applications (VS-GAMES), 2011 Third International Conference on, IEEE, 2011, pp. 47–54.

[21] Cheung, Alvin and Kamil, Shoaib and Solar-Lezama, Armando, Bridging the gap between general-purpose and domain-specific compilers with synthesis, in: LIPIcs-Leibniz International Proceedings in Informatics, Vol. 32, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[22] M. e. a. Best, Searching for concurrent design patterns in video games, in: Euro-Par 2009 Parallel Processing, Springer, 2009, pp. 912–923.

[23] A. Reed, Creating Interactive Fiction with Inform 7, Cengage Learning, 2010.

[24] J. Mallett, M. Lefler, Zillions of games, `www.zillions-of-games.com` (1998).

[25] G. Maggiore, M. Bugliesi, R. Orsini, Monadic scripting in f# for computer games, in: TTSS 115th International Workshop on Harnessing Theories for Tool Support in Software, 2011, p. 35.

[26] N. Tabareau, I. Figueroa, É. Tanter, A typed monadic embedding of aspects, in: Proceedings of the 12th annual international conference on Aspect-oriented software development, ACM, 2013, pp. 171–184.

[27] Galcon, `https://www.galcon.com/`.

[28] A. Snyder, Encapsulation and inheritance in object-oriented programming languages, in: ACM Sigplan Notices, Vol. 21, ACM, 1986, pp. 38–45.

[29] J. Eder, G. Kappel, M. Schrefl, Coupling and cohesion in object-oriented systems, Tech. rep., University of Klagenfurt, Austria (1994).

[30] W. White, A. Demers, C. Koch, J. Gehrke, R. Rajagopalan, Scaling games to epic proportions, in: Proceedings of the 2007 ACM SIGMOD international conference on Management of data, ACM, 2007, pp. 31–42.

[31] J. Courtney, Using Ant Colonization Optimization to Control Difficulty in Video Game AI., http://dc.etsu.edu/honors/147, undergraduate Honors Theses (2010).

[32] C. Granberg, David Perry on game design: a brainstorming toolbox, Cengage Learning, 2014.

[33] M. Wolf, The video game explosion: a history from PONG to Playstation and beyond, ABC-CLIO, 2008.

[34] M. Claypool, K. Claypool, Latency and player actions in online games, Communications of the ACM 49 (11) (2006) 40–45.

[35] E. Kaiser, W. Feng, PlayerRating: a reputation system for multiplayer online games, in: Proceedings of the 8th Annual Workshop on Network and Systems Support for Games, IEEE Press, 2009, p. 8.

[36] J. Smed, T. Kaukoranta, H. Hakonen, Aspects of networking in multiplayer computer games, The Electronic Library 20 (2) (2002) 87–97.

[37] S. Bhatti, E. Brady, K. Hammond, J. McKinna, Domain specific languages (DSLs) for network protocols, in: International Workshop on Next Generation Network Architecture (NGNA 2009), 2009.

[38] F. Glinka, A. Ploß, J. Müller-lden, S. Gorlatch, RTF: a real-time framework for developing scalable multiplayer online games, in: Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games, ACM, 2007, pp. 81–86.

[39] G. Russell, A. Donaldson, P. Sheppard, Tackling online game development problems with a novel network scripting language, in: Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games, ACM, 2008, pp. 85–90.

[40] C. Tismer, Stackless Python, http://www.stackless.com/.

[41] J. Armstrong, R. Virding, C. Wikström, M. Williams, Concurrent programming in ERLANG, Prentice Hall, 1993.

[42] Unreal Technology, https://www.unrealengine.com/.

[43] Unity Game Engine, http://unity3d.com.

[44] B. Ban, JGroups - A Toolkit for Reliable Multicast Communication, `http://www.jgroups.org/index.html` (2002).

[45] F. Di Giacomo, M. Abbadi, A. Cortesi, P. Spronck, G. Maggiore, Building game scripting DSL's with the Metacasanova metacompiler, Springer International Publishing, 2016, pp. 231–242.