High performance encapsulation in Casanova 2

Mohamed Abbadi, Francesco Di Giacomo,
Agostino Cortesi

Ca'Foscari University
Venice, Italy
mohamed.abbadi,francesco.digiacomo,cortesi@unive.it

Pieter Spronck

Tilburg University
Tilburg, The Netherlands
p.spronck@uvt.nl

Costantini Giulia,
Giuseppe Maggiore

Hogeschool Rotterdam
Rotterdam, The Netherlands
costg,maggg@hr.nl

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.**

I. Introduction

The video games industry is an ever growing sector with sales surpassing 20 billion dollars in 2014 [2]. Video games are not only built for entertainment purposes, but they are also used for Edutainment, Higher Education, Health Care, Corporate, Military, Research, and other [1], [12]. These so-called *serious games* usually do not enjoy the budgets available in the entertainment industry [15]. 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 [11]), but these tend to produce simple games in a specific genre, that are hard to customize. Therefore, most game developers rely on a general-purpose language (GPL) to create games [9]. Such languages, however, lack the domain-specific abstractions and optimizations of games [13],

[16], leading to highly complex code that is expensive to maintain.

Software development techniques have been studied in the past years to improve software maintainability and tackle software complexity. Encapsulation, which is a software development technique that consists of isolating a set of data and operations on those data within a module and providing precise specifications for the module [8], is a typical technique used to increase code readability and maintainability [5].

Games feature many small entities that interact with each other. Encapsulation forces those entities to interact through specific interfaces. Therefore, when calling methods of the interfaces, overhead is added due to dynamic dispatching. Such overhead ultimately affects the performance of games at runtime negatively. Performance is of high importance for games, since it is strictly connected to game smoothness, i.e., to the game's framerate, where a frame consists of a complete update of all the entities present in the game. Smoothness strongly influences the perceived quality of a game [4].

Our goal is to develop techniques for taming the complexity of games by means of encapsulation, increasing code readability and maintainability, without losing performance. In this paper we present a solution to the loss of performance in encapsulated programs. We will show a domain specific language for games, named "Casanova 2", which allows developers to write high quality games at reduced development costs. Our solution allows developers to write encapsulated code which, through extensive automated optimization, turns source code into high-performance executable code.

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 II). We introduce our idea for dealing with encapsulation without losing performance (Section III). We propose a specific implementation, with corresponding semantics, within the Casanova 2 language (Section IV). We then evaluate the effectiveness of our approach in terms of performance and compactness (Section V), and round off with conclusions (Section VI).

II. ENCAPSULATION IN GAMES

In this section we introduce a short example to explain the problem of encapsulation in games. We then discuss the advantages and disadvantages of using encapsulation when designing a game.

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 [3]). 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 point as references to `Planets`, and (*ii*) a list of `Fleets` traveling 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.

According to the definition of encapsulation, data and operations on them must be isolated within a module and a precise interface must be provided. Moreover, each entity is responsible for updating its own fields in such a way that it maintains its own invariant.

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: i) **Planet**: Takes the enemy fleets traveling along its incoming routes which are close to the planet, and moves them into the attacking fleets list; ii) **Route**: Removes the traveling fleets which have been placed in the attacking fleets of the destination planet from the list of traveling fleets.

```
class Route
    Planet Start, Planet End,
    List<Fleet> TravellingFleets,
    Player Owner
    void Update()
      foreach fleet in TravellingFleets
        if End.AttackingFleets.Contains(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)
```

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 [14], 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 [7].

As far as *performance* is concerned, in the encapsulated version, the planet queries the game state to obtain all routes which endpoints are the planet itself, and for every route selects the enemy traveling 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 which are contained in both lists from the traveling fleets. If we consider a scenario containing m planets, n routes, and at most k traveling 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.

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.

III. 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.

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 traveling 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 which 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.

Optimizing temporal/local predicates: If we take into consideration the fact that predicates belong to (potentially hundreds or thousands) entities in a simulation that exhibit similar behaviors (ships, bullets, asteroids, etc.) [6], 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* collection, and (*ii*) activate 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.

Language 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 2, which is a game development oriented language, ensures that variables are only changed through specific statements; this makes it possible for the Casanova 2 compiler to identify patterns in code which 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 data structures. In the next session we present the compiler architecture and the transformation rules.

IV. IMPLEMENTATION DETAILS

In this section we introduce the syntax of the Casanova 2 language and show how to select the predicates and the associated blocks of code which can be optimized.

Most games represent simulations of some sort. A property of simulations is a certain *temporal locality* of behaviors [6]. 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 optimize those predicates that exhibit temporal locality (the selection is based on manual annotation).

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 specific data-structure) to suspend and wake up with little overhead. This is schematically shown in Figure 1.
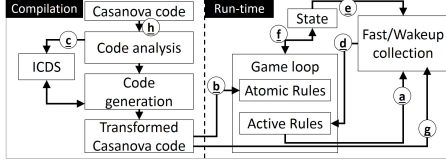

Fig. 1: System Configuration

Casanova overview: 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 [10]. Casanova 2 (which we use) is a recent iteration of the original Casanova, which does not introduce changes to syntax or semantics.

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 *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
```

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 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: i) A *suspendable statement* is either a `wait` or a `yield`; ii) 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. iii) An *atomic rule* is a rule which 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 to build the ICDS (Algorithm 2). 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.

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

function ATOMIC(p)
  E is the set of entities.
  $DFA \leftarrow \emptyset$
  for $e \in E$ **do**
    R is the set of rules in e
    for $r \in R$ **do**
      if r is an atomic rule **then**
        for $f \in r.domain$ **do**
          $DFA \cup \{(e, f)\}$
  $D \leftarrow$ set of $(entity, field)$ in the predicate p.
  return $\exists x \in D \; : \; x \in DFA$.

Algorithm 2 ICDS construction

function BUILDICDS( )
  $ICDS \leftarrow \emptyset$
  E is the set of entities.
  for $e \in E$ **do**
    R is the set of rules in e
    for $r \in R$ **do**
      if r is an ISR **then**
        p is the first interesting condition of r
        if not ATOMIC(p) **then**
          $ICDS \cup \{(e, r.index, r.domain, p)\}$
  return $ICDS$

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.

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 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 shape 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, 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)`.

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: i) **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 of which rules are targeted by an IC, we add the pair made of the entity instance and each targeted rule as a value in its DEPs; ii) **On destroy**: when an instance which appears either as a value or a key in one of DEPs, we remove all the occurrences of the instance in DEPs; iii) **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 that contains per entity all the suspended rules that are run during a game iteration.

Rules in Casanova are translated at compile time into a series of switches without nesting within functions which 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)).

Query interpretation: We transform a query into semantically equivalent code where every entity appearing in the `from` expression (*source*) adds or removes itself from an index stored in the entity containing the query (*target*). We add or remove a source entity in the target index only if the condition is `true`. 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.

V. Evaluation

In this section we evaluate the performance of our approach. A comparison on the same Casanova game code between the not optimized implementation and the optimized one, and an implementation in C#, will be

shown and discussed in terms of run time performance and code complexity.

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 random 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.

Performance evaluation: Table II 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 with respect to the non-optimized code. Using MonoGame the optimization is linearly faster. The difference is due to the implementation of the underlying frameworks.

TABLE I: Code lines comparison

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

TABLE II: Running time comparison

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

Code size evaluation: Table I shows the code length for each implementation. Casanova 2 game code needs about half the lines of code compared to the idiomatic C# implementation. The intermediate code that the Casanova 2 compiler creates (which is C# code) is considerably longer due to the presence of the 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.

VI. CONCLUSIONS

Game developers often have to choose between maintainability of their code and speed of execution, a choice which 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. We therefore conclude that our approach allows game developers to write high-performance code without losing maintainability.

REFERENCES

[1] CMP Media Game Developers Conference, 2004.
[2] Essential facts about the computer and video game industry. http://www.theesa.com/wp-content/uploads/2015/04/ESA-Essential-Facts-2015.pdf, 2015.
[3] Galcon. https://www.galcon.com/, 2015.
[4] M. Claypool and K. Claypool. Perspectives, frame rates and resolutions: it's all in the game. In *International Conference on Foundations of Digital Games*. ACM, 2009.
[5] E. Collar Jr and R. Valerdi. Role of software readability on software development cost. 2006.
[6] J. Courtney. Using ant colonization optimization to control difficulty in video game ai. *Undergraduate Honors Theses*, 2010.
[7] J. Eder, G. Kappel, and M. Schrefl. Coupling and cohesion in object-oriented systems. *Technical Report, University of Klagenfurt, Austria*, 1994.
[8] ISO/IEC/IEEE. ISO/IEC/IEEE 24765 - Systems and software engineering - Vocabulary. Technical report, 2010.
[9] M. Lewis and J. Jacobson. Game engines. *Communications of the ACM*, 2002.
[10] G. Maggiore. Casanova: a language for game development. 2013.
[11] P. Petridis, I. Dunwell, S. De Freitas, and D. Panzoli. An engine selection methodology for high fidelity serious games. In *Games and Virtual Worlds for Serious Applications*. IEEE, 2010.
[12] M. Prensky. Computer games and learning: Digital game-based learning. *Handbook of computer game studies*, 2005.
[13] K. Rocki, M. Burtscher, and R. Suda. The future of accelerator programming: Abstraction, performance or can we have both? Symposium on Applied Computing. ACM, 2014.
[14] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *ACM Sigplan Notices*, 1986.
[15] A. J. Stapleton. Serious games: Serious opportunities. In *Australian Game Developers Conference, Academic Summit, Melbourne*, 2004.
[16] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *Transactions on Embedded Computing Systems*, 2014.