High performance encapsulation in Casanova 2

Abstract

Modern video games require high-performance components, which are usually implemented by scratch using general purpose programming languages (such as C++ or C#). Therefore programmers are forced to write the optimization code by hand, affecting readability and maintainability of the program. In this paper we propose a solution to the difficulties arising from manually implementing optimizations in the code and, at the same time, writing a readable and maintainable program. We introduce *Casanova*, a language oriented to game development, and we present the implementation of its compiler which automates the process of optimizing the code. We show that our compiler produces a code which is readable, maintainable, and high-performance with respect to other implementations in languages commonly used by video game industry.

1 Introduction

Video games industry is an ever growing sector with sales surpassing 100 million $ in 2014 [4]. Video games are not only built for entertainment purposes but they are also used by schools, government agencies, military, and research. These "serious" games are not allocated the development budgets available for the entertainment industry, thus the developers are interested in tools capable of overcoming the difficulties deriving from the complexity of games, and the long development time required to build their product.

Video games are complex and composed of several inter-operating components, which accomplish different and coordinated tasks, such as drawing a game object, running the physics simulation of bodies, and the artificial intelligence moving non-playable characters. These components are periodically activated in turn to update the game state and draw the scene. The rate at which these components update the game state can seriously affect the quality of the game, as the player becomes less efficient in performing precision tasks (such as jumping across a chasm, clicking a small interface button, etc.) and also perceives a less overall quality in the play experience [2, 3]. Computer games are still mostly developed using general purpose programming languages, such as C++ and C#. Given their general purpose nature, these languages are not ideal to define common patterns used in games, thus forcing the programmers to implement them by hand. Manual implementations seek high-performance but lose readability of the code, since they alter the program structure by adding auxiliary data structures. These data structures introduce additional dependencies among the game elements to run the optimization, which affect the understandability and complicate the program modification [?]. A solution is using the encapsulation design pattern to increase readability but is low-performance [?], thus developers usually discard this design pattern for a high-performance solution.

In this paper we present a solution to the loss of performance in encapsulated programs by using a domain specific language, Casanova, that allows developers to write readable and maintainable code and, at the same time, relives them from writing optimizations by hand to gain high-performance.

In Section 2 we introduce a sample of a game implemented both with encapsulated code and a faster implementation which breaks encapsulation. We will then discuss the complexity of both solutions. In Section 3 we explain the idea behind our transformation system. In Section 4 we build the transformation system inside Casanova compiler. In Section 5 we show a comparison between our optimized encapsulated code and a not encapsulated implementation.

2 Encpasulation in games

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

2.1 Running example

Let us consider a game consisting of a set of planets linked together by routes. In what follows we call *frame* a single update cycle of the game data structures. The player can move fleets from its 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 around the planet. In our example we will 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` travelling through the route. The `Planet` is represented by a data structure containing (*i*) a list of defending `Fleets`, (*ii*) a list of attacking `Fleets`, and (*iii*) an `Owner`. We also assume that each fleet has an owner as well. Each data structure contains a method called `Update` which updates the state of its class 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.

2.2 Comparing design techniques

According to the definition of encapsulation it is required that data and operations on them must be isolated within a module and a precise interface is provided [?].

In our example the modules are `Planet` and `Route` class defined above, data are their fields, and operations are the following:

- **Planet:** Take the enemy fleets travelling along its incoming routes which are close to the planet, and move them into the attacking fleets list,

- **Route:** Remove the travelling fleets which have placed in the attacking fleets of the destination planet from the list of travelling fleets.

```
class Route
  private Planet Start, Planet End,
          List<Fleet> TravellingFleets,
          Player Owner

  public void Update()
    foreach fleet in TravellingFleets
      if End.AttackingFleets.Contains(fleet)
        this.TravellingFleets.Remove(fleet)

class Planet
  private List<Fleet> DefendingFleets,
          List<Fleet> AttackingFleets

  public void Update()
    foreach route in GetState().Routes
      if route.End = this then
        foreach fleet in route.TravellingFleets
          if || fleet.Position − this.Position || < δ &&
              fleet.Owner != this.Owner then
                this.AttackingFleets.Add(fleet)
```

An alternative design not using encapsulation allows the route to move directly the fleets close to the destination planet 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. Below we provide the pseudo-code for the update functions implemented with referential opaqueness [1].

```
class Route
  private Planet Start, Planet End,
          List<Fleet> TravellingFleets

  public void Update()
    foreach fleet in this.TravellingFleets
      if || fleet.Position − End.Position || < δ &&
         fleet.Owner != End.Owner then
        this.TravellingFleets.Remove(fleet)
        End.AttackingFleets.Add(fleet)
```

2.3 Discussion

In the fleet sample a programmer is left with the choice of either using the paradigm of encapsulation which improves the understandability of programs and ease their modification [?], or breaking encapsulation by writing directly into the planet fields from an external class, which, as we will show below, is more efficient.

In the encapsulated version the planet queries the game state to obtain all the travelling fleets to retrieve the links whose endpoints are the planet itself. 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 travelling fleets. If we consider a scenario containing m planets, n routes, and at most k travelling fleets per link, each planet should check the distance condition for $O(nk)$ ships, thus the overall complexity is $O(mnk)$.

[1] The code of the planet is missing because the fleet transfer is managed entirely by the `Route` class. In this case we assume that either the fields of `Planet` are public or a setter method exists.

On the other hand, we can break encapsulation in such way that each link checks the distance for a maximum of k ships and then directly move those close to the planet, and this is done in time $O(nk)$.

In the following section we define the idea behind a transformation from encapsulated code to a high-performance implementation.

3 Optimizing encapsulation with code transformations

In Section 2 we addressed the problem of the trade off between software engineering compliant implementations, such as the encapsulation, and performance. In this section we introduce the idea behind a map between encapsulated programs and a semantically equivalent and efficient implementation.

3.1 Optimization overview

In the example we saw that the main drawback of encapsulation is that each planet has to check all the fleets to see if they are close enough and to move them in the list of attacking fleets. An optimization can be achieved by maintaining an index `FleetIndex` in `Planet` containing a list of `Fleets` which satisfy the predicate in the query (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 inside `FleetIndex` by the `Route` which stores a list of travelling fleets. When `FleetIndex` changes, it notifies it to `Planet`, so that it can update `AttackingFleets`.

We can generalize this situation saying that encapsulation suffers the loss of performance whenever an object B needs to update one of its fields depending on a property of another object A or one of its fields. The object B stores an index I_A which is used to keep track of the objects satisfying the predicate. The object A has a reference to B and it is tasked to update the index I_A of B. B checks I_A every time it needs to interact with the elements of A satisfying the predicate.

3.2 Language level integration

This generalization breaks encapsulation, since the entity A is writing an index which is stored in B. This process can be integrated at 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. Unfortunately general purpose programming languages do not impose a strict discipline on writing variables and thus we claim they are not suitable for our purpose. We chose Casanova 2, which is a game development oriented language, where variable writing is disciplined and possible only through specific statements.

The compiler will apply transformations to the code that preserve the program semantics and optimize the encapsulated implementation by creating and maintaining the required indices. In this way the code written by the programmer will benefit of the clearness and maintainability of an encapsulated program without neither suffering the loss of performance nor requiring to break encapsulation to manage the optimization data structures.

In Section 4 we present the compiler architecture and the transformation rules.

4 Encapsulation optimization in Casanova compiler

5 Evaluation

References

[1] R.P.L. Buse and W.R. Weimer. Learning a metric for code readability. *IEEE Transactions on Software Engineering*, 36(4):546–558, jul. 2010.

[2] Kajal T. Claypool and Mark Claypool. On frame rate and player performance in first person shooter games. *Multimedia Syst.*, 13(1):3–17, 2007.

[3] Mark Claypool, Kajal Claypool, and Feissal Damaa. The effects of frame rate and resolution on users playing first person shooter games. *Proc. SPIE*, 6071:607101–607101–11, 2006.

[4] Rob van der Meulen, Janessa Rivera. Gartner says worldwide video game market to total $93 billion in 2013. `http://www.gartner.com/newsroom/id/2614915`.

[5] Christopher Strachey. Fundamental concepts in programming languages. *Higher Order Symbol. Comput.*, 13(1-2):11–49, April 2000.