Università
Ca'Foscari
Venezia

Corso di Dottorato di ricerca

in Informatica

ciclo 29

Tesi di Ricerca

Casanova 2: a domain specific language for general game development

SSD: INF/01

Coordinatore del Dottorato
ch. prof. Riccardo Focardi

Supervisore
ch. prof. Agostino Cortesi

Dottorando
Mohamed Abbadi
823145

Contents

Abstract

In today's society, the pervasiveness and sales of video games is at an all-time high. Video games are used in a variety of application scenarios, from pure entertainment to supporting research, raising social awareness, and training. Video games are no longer developed only by professional programmers, but also by experts in other domains. This has made the problems surrounding the process of game development increasingly evident. One such problem is the lack of a clear methodology for defining video games, supported by user-friendly tools. Indeed, the available tools for making video games are either too specific or too general. When too specific, the abstractions provided by the tool are so poor that only few game genres are expressible. When too general the abstractions provided by the tool are so generic that even expressing simple domain concepts requires a lot of effort.

These problems lead to the process of developing video games being a costly one, in terms of time, money, and necessary knowledge. Such costs negatively affect the development process, and may even lead to the impossibility to develop certain games. When a solution is offered that reduces the cost of game development, this will benefit in particular the developers for whom game development is not their main job.

This thesis starts by analyzing the process of making a video game, and examines the available tools for making them. It then proposes a solution to the high costs of making games. This solution comes in the shape of a programming language that is exclusively focused on the domain of video games. This language, which we call Casanova 2 (inspired by its predecessor language Casanova, with which it shares goals and philosophy), is designed to offer abstractions built around the typical aspects of video games. Casanova 2 is not bound to any video game genre. Due to the specificity of the domain of game development, and the strong requirements it brings with it, the compiler behind the Casanova 2 language is able to apply code analysis. Together with a series of optimization layers, it is able to turn complex domain code into a highly-performant executable. Casanova 2 comes with a series of advantages such as embedded networking, and high-performance encapsulation support, which positively affects the production of games.

The thesis evaluates Casanova 2 by comparing it with representative languages, that are often used for video games, on expressiveness, compactness, speed, ease-of-development, and maintainability. It demonstrates that Casanova 2 is either equivalent to or outranks all competitors in these respects. This warrants the conclusion that Casanova 2 achieves its goal of offering a game development language that can be successfully used by a wide variety of developers to build video games.

Chapter 1

Introduction

In this chapter we discuss traditional games and video games. We discuss the requirements that define traditional games and show that video games fulfill such requirements. We show how by means of digital media new games design opportunities become possible. We discuss the difficulties arising from the development of video games, and to what extent these difficulties affect video game developers. This discussion will eventually lead us to the definition of the problem statement and research questions of this thesis. We conclude this chapter with the thesis outline.

1.1 Traditional games

Before games became digital, they used to be played either indoor, by means of physical objects, such as chess, or outdoors, as in sports. We refer to these kinds of non digital games as "traditional games". In Homo Ludens [55], Johan Huizinga states that games (or playing) are at the foundation of many cultures and societies, as games are a universal part of the human experience since ancient times up until now. We can find traces of games in many cultures from the past. Among the oldest games we find: the Royal Game of Ur [92], a board game from the First Dynasty of Ur, dated about 2500 BC; Senet (or Senat) [85], a board game from predynastic and ancient Egypt, dated about 3500 BC; and Polo [27], a sport game designed to develop military skills, for which oldest records were discovered in Persia, dated about 600 BC. Nowadays, pervasiveness of games is at an all-time hight; games are played by different kinds of players regardless of their social status, age, gender, etc.

Among these games we find traditional games such as board games, sport games, and card games. In the following, we discuss the fundamental aspects that are common to all traditional games.

1.1.1 Key ingredients of traditional games

A reason for the success of traditional games is associated with their ability to involve people regardless of age or gender. Such success is based on three essential *ingredients*: goals, challenges, rules [88]:

1. *Goals*: the desired final results that the player plans to achieve. According to Chris Crawford [32] a game without a goal should be considered a toy rather than a game;

2. *Challenges*: obstacles in a game for the player to overcome, intended to make a game more difficult, and interesting, or to extend the total play time;

3. *Rules* define the dynamics of a game. They can be *active* or *passive*.

- Active: any part of the rules system of a game that regulates interaction that takes place in a game at any time, be it general or specific;
- Passive: constraints over the game dynamics. Such constraints can also be used in games to make players more comfortable with playing the game, or to speed up the learning process of the game mechanics. For example: by using classic physics as general rule in a game, players are not required to learn how to move in the game world; by using specific colors or uniforms to identify the enemy faction, the game looks less chaotic, etc.

By manipulating the above ingredients we can achieve different *flavors*. Such flavors can be combined in order to implement the so-called game *genres* [103]. Game genres are used in order to reach different targets of players [49]. These targets may vary depending on different aspects, such as the demographics of players, or the desired result of the game experience. Typical genres of traditional games are billiards, board games, card games, etc.

1.1.2 Functions/goals

Every game comes with a series of goals. Such goals can be either self motivated, or provided/enforced externally (by an instructor for example). When self-motivated, a goal can be the result of a logical or biological need, or a social factor. For example, the socializing aspect of supporting a football team.

When the goals are provided externally, the game can become a means of accomplishing real-life, beneficial objectives, for example set up by an organization. In order to achieve such goals, a player is subject to a series of situations that, together with bringing him closer to the goal, become experiences from which the player can learn new skills or abilities. Games belonging to this category are typically referred to as *serious games*. In this thesis we will focus on the development process of serious games, since serious games are important in terms of social impact.

In all games, regardless of whether they are self motivated or not, and serious or not, we can notice a common factor that is the "fuel" that makes people play them; this fuel is called *fun*. Without fun it would be very difficult (maybe even impossible) to manage to achieve the original desired function of a game. Indeed, according to [57], fun is a characteristic that every game must have in order to be defined as such.

About serious games

Serious games are the result of a careful mixture of the ingredients introduced in Section 1.1.1. The function of serious games is different from that of other kinds of games, since they are not only meant for entertaining, but also for educating, raising awareness, etc. In his book Abt [5] gives a good definition of what serious games are: "*Games may be played seriously or casually. We are concerned with serious games in the sense that these games have an explicit and carefully thought-out educational purpose and are not intended to be played primarily for amusement. This does not mean that serious games are not, or should not, be entertaining*".

We find traces of serious games in the past. Polo was used by Persians to teach their soldiers how to fight while riding a horse. Nowadays serious games are adopted by several organizations to educate or train their members on subjects such as politics, military skills, etc. [35]

Serious games really became a class of games on their own with the advent of video games, which are discussed next.

1.2 Video games

Next to the concept of traditional game, in recent times the concept of *digital game* has appeared. A digital game, or more commonly called *video game*, is a game where a user is required to interact with a user interface presented and handled by a digital computer.

Video games first appeared around the mid 50's when the first computers were created. Among all games, *OXO*, also known as Noughts and Crosses, is generally considered to be the very first video game in history. Also known as tic-tac-toe, OXO was the first video game that supported input and output devices: a phone dial (each number corresponded to a cell of the game grid) was used for the input, and a CRT monitor for the output.

1.2.1 What is a video game?

One can ask the question whether the only difference between games and video games is that video games require the use of digital media. In the most general sense, the answer is "yes". However, a digital computer (which is mainly a series of physical devices controlled by software) opens up a series of new opportunities that traditional games cannot provide. This is possible due to the fact that the hardware that runs a video game can be programmed. A video game always consists of a series of instructions that the computer hardware processes sequentially, in order to provide a desired experience. The range of achievable experiences is becoming increasingly immersive as new devices are entering the market. For instance, augmented reality devices involve more of the player's senses more pervasively, and are therefore able to achieve deeper levels of immersion.

In the following we show why video games can be considered actual games; we do so by showing that video games share the same ingredients as traditional games.

- *Goals*: just as with traditional games, all video games have a goal that entails the final result that the player plans to achieve.
- *Challenges*: just as with traditional games, with video games there are obstacles in a game meant to regulate game aspects such as difficulty, or play time.
- *Rules*: just as with traditional games, video games are based on rules. The main difference it that the rules of video games can be atomized.
  - *Active*: in a video game, every game element can be programmed as to automatically react to a series of user inputs. Reactivity is one the most important aspects of video games: by mapping every action of the player to a specific reaction, a feedback loop chain is set up that eventually will enhance the player's experience, as the player will feel as if he or she is an active part of the game itself
  - *Passive*: a video game can be programmed so to provide a series of constraints that limit the range of possible actions of the player. Rules can also be used to make the player feel more comfortable with the game (for example, by simulating natural gravity the physics feel comfortable to a player, and by using particular colors and clothing options for enemies, they become easily identifiable to the player), or to force the player to follow the story line, so as to achieve eventually the final goal of the game.

Video games also come with a series of advantages that cannot be found in other kinds of games. These advantages are typical of video games and are possible because of the adoption of digital media. Some of these advantages are:

- Absence of physical constraints: a video game allows the definition of worlds or objects that could not exist in real life. For example, a video game might feature a world in which entities are all subject to a different gravity than the one we find on earth, or a video game might feature enormous galaxies made of billions of star systems.
- Control over the flow of time: in a video game time can be programmed and controlled/processed by the computer. The flow of time in a game might be dynamically adjusted in order to provide players with different kinds of experiences. A player has almost no control over the timings of a game (unless the game rules allow it). As result, during the game, time can speed up, slow down, or even pause.
- Visual effects: the rendering components of a computer make it possible to visualize on a 2D screen the elements of a video game (including their states). Moreover, visual effects can be programmed to increase the game's appeal, or to reinforce the player's involvement.
- Artificial intelligence: a player can be assisted by, or play against, an *artificial intelligence* (AI). Typically, this AI, which is previously programmed by the developers of a video game, is subject to the game rules and dynamics, and in many cases is programmed to behave almost as a "real" player.
- Assisted learning curve: a video game can provide tutorials before, or during, a game. These tutorials, which are typically programmed, are used to introduce a player to the main aspects of a video game and to make him focus immediately on the important aspects of a the game. It is often the case that these tutorials are incremental and come slowly during the game, as introducing too many aspects of a complex game might confuse the player.

Just like with traditional games, video games genres are defined by particular implementations of the above aspects. For example, consider an economy city builder. An economy city builder game, such as SimCity, a player is tasked to manage in real-time the micro and macro economy of a city in order to make it prosper and grow. In such a game we would typically have:

- big scenarios (possible only due to the absence of physical constraints) full of cities, each with its own economy and dynamics, possibly controlled by other human players or AIs,
- control over the flow of time in the hands of the player, who can pause the game, slow it down, or speed it up,
- average rendering effects, as the logic engine (the city simulator) is the real selling point of games belonging to this category,
- an AI that is specialized on automating processes of macro and micro economy of the city,
- an assisted learning feature that introduces incrementally the important elements of the game.

Such a game could not exist in the form of a traditional game (without large modifications), because all the above aspects are difficult, if not impossible, to reproduce with non-digital components. For example a video game featuring gigantic artificial cities, made of millions of citizens and objects, would require enormous spaces, and the manual of such game would come with hundreds of pages in order to describe all possible game features and mechanics.

Video game genres differ from each other a lot. Among such genres genres we find: *Platform games*, such as Donkey Kong, Super Mario Bros, Jumping Flash!; *First person shooter games* (FPS's), such as Wolfenstein 3D, Call of Duty, Half Life; *Role-playing video games* (RPG's), such as Diablo, Dungeon Siege, Baldur's Gate; *Real-time strategy games* (RTS's), such as Age of Empires, Warcraft, Startcraft;

Sport games, such as FIFA, Pro Evolution Soccer; *Music games*, such as Rock Band, Guitar Hero, Sing Star; Massive multiplayer online role-playing games (MMORPG's), such as Second Life, Ingress, The Elder Scrolls Online, Final Fantasy XI, EVE Online; etc. Of course, some of these genres are somehow re-arrangements/evolutions of the genres of traditional games introduced in Section 1.1.1.

1.2.2 Business impact

It took several years for video games to become a global phenomenon, since in the beginning video games were mostly used by the scientific community for experiments. However, slowly, video games started to be used also for entertainment purposes, in particular when console games started to become popular. Everything changed in 1972 when Atari presented the game *Pong*. Pong is generally considered the first official video game in history. Pong helped establish the video game industry with great sales. Indeed, after the great success of Pong, many companies started to copy it and to present new versions of it. This pushed Atari even more to produce more innovative games in order to beat the competitors, and so the modern game industry was born.

This continuous exploration and competition pushed video game companies to study and develop new kinds of video games; part of this exploration was also justified by the advances of computer hardware (one could even say that hardware advances in personal computers are also partially caused by the popularity of video games). With more powerful hardware developers could study and develop new techniques such as better visual effects, or define more complex artificial intelligence.

As a result, video games have grown to the point that their sales have surpassed those of music and movies (together) [2]. Mobiles video games have contributed to this big success: in 2017 alone sales of mobile games are predicted to exceed 100 billion dollars worldwide [100].

This success can only strengthen the fact that our initial statement (about the relevance of games as a social phenomenon) cannot be ignored. A remarkable example of this is *Pokemon GO*, produced by Nintendo. Pokemon GO is a game that requires players to physically move across different physical locations in order to play the game and thus capture Pokemon.

In Europe it took just a few months to get it to be installed on millions of devices without any sort of advertisement. Pokemon Go has become so famous and is played so much that governments limited its usage in many public areas, such as museums, religious sites, or hospitals, as players disrupt, or obstruct the intended activities of such public areas.

1.2.3 Functions/purposes

As stated in Section 1.1.2, video games are mainly played for fun. The main revenue of video games comes indeed from the entertainment sector, involving games such as Tetris, Wii Sports, Grand Theft Auto 5, or Super Mario Bros. Entertainment video games are generally sold more than traditional games, mainly due to the advantages offered by digital media. Among those advantages we find: complex story lines that can take days, or even weeks, to be finished; large worlds to discover; leaderboards where a player can compare his/her performance with the performances of other players playing the same game; relationships that can be established with other players playing the same game; multiplayer with players playing the same game in different places across the Internet; etc.

Serious games

Video games are also used in serious scenarios. Due to the new advantages offered by digital media, serious games as a whole are experiencing a new phase of their historical development. Nowadays, serious video games are used in many different scenarios such as research, education, healthcare, defence, art, culture, religion, corporate training and advertising [35, 21, 97].

Besides being successful, because of their realism, and thus for their ability to connect with real life challenges, nowadays the success of serious games is also pushed by the adoption of the digital medium, which can reinforce the overall gaming experience with elements such as pleasant visualizations, audios, automated reward systems, and complex rules simulating real contexts, and thus the realism of the serious game itself.

Serious games have been used also as frameworks for various kinds of scientific research, since in research testing and building running examples are of much importance to validate experiments. Not surprisingly, the first video games were meant to illustrate scientific results, or research experiments: for example serious video games have been employed to study possible ways of interaction between human and machine (*OXO* was the result of a computer science thesis in the now established research field of *human-computer interaction*[35]).

1.3 Building video games

After this brief and compact introduction to traditional games and video games, we can now focus our attention on the subject of this thesis, in particular, on the building of video games. We will try to understand the complexity behind this process in order to figure out the fundamental aspects that define it.

In the following we start with considering and understanding the fundamental processes underlying the main process of developing a video game. We do so by considering the main professional roles involved in the process.

1.3.1 On the process of making a video game

The process of developing a video game typically involves many professional roles with different expertises. In the following we present such professional roles, which we group under three main categories: designers, artists, and programmers [15]. A visual representation is also provided (see Figure 1.1), that explicitly shows the interactions between the following professional roles.

- **Designers** are the initial project coordinator when a game is created. More precisely, designers are responsible for all those components that make up a game, without actually creating any of these components. Designers receive the project of a game to design from a **client**. Typically, the request of a game to design follows the trend of the market. Indeed, the client is in continuous contact with the target **users** to understand the market's necessities. Indeed, if there is a huge request for a feature, then the client will forward this demand to the designers during the commission phase of the game.

  Typically, designers work very closely to the programmer, as they have to continuously test the game, with the so called **testers** in order to determine the effect of the design choices, and to provide feedback on how to improve the game. Testers, are a group of selected users, but also professionals, tasked to play the game in all its development phases and to provide feedback to designers, and programmers, about the functional and non functional aspects of the game.

- **Artists** deal with everything related to the game content: visual, audio, etc. They work very closely with designers, as the **contents** they make have to be as similar as possible to those envisioned by the designers.

- **Programmers** are tasked to implement the designers' choices into actual machine instructions and include the artists' contents in the game. Since these tasks cover different aspects of a game, they require different kinds of expertises (typically, provided by different programmers). The

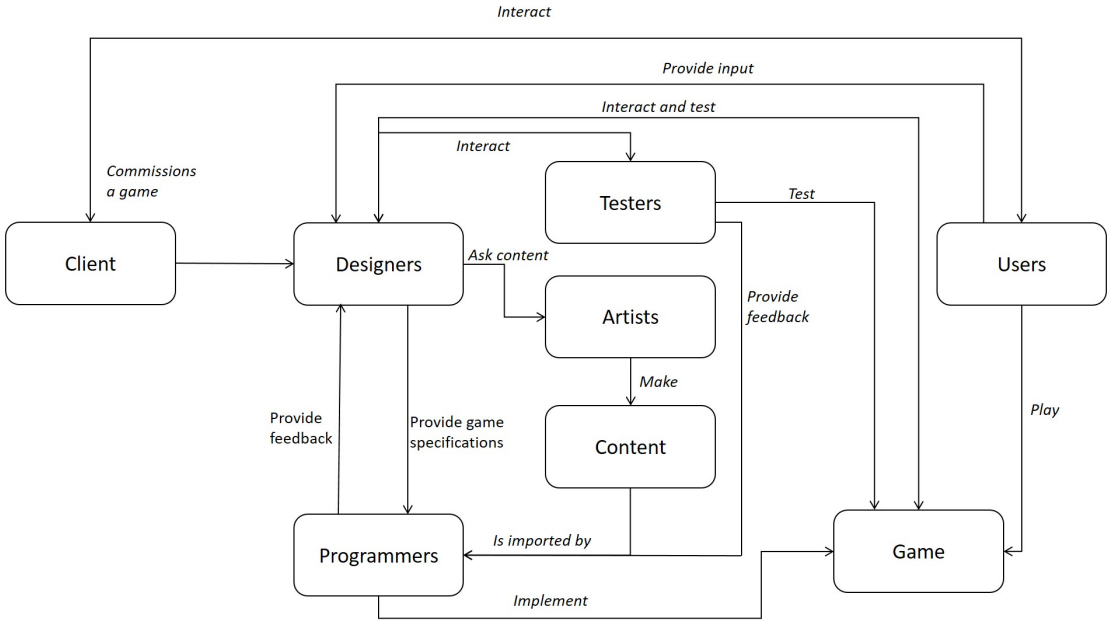


Figure 1.1: A graph describing the interactions among the professional roles involved in the process of developing a video game.

output of this process is the actual **game**, which eventually will be played by the target users of the game.

- *Contents integration*: for this task a programmer is supposed to provide support to the team in order to make the contents generated by artists accessible within the game, for example by providing a custom importer and processor for the contents;
- *Rendering*: for this task a programmer is supposed to write the code necessary to display all entities of a game and the visual effects to apply to them, for example by means of custom shaders;
- *Logic*: for this task a programmer is supposed to provide the code necessary to implement the aspects that are necessary to express all the game dynamics: AI, physics, networking, scripts, I/O controllers, etc. Typically, this task includes also both the codes provided by the previous two tasks (content integration and rendering).

Every task mentioned above comes with costs that need to be considered while developing a video game. These costs become even more incisive if the tools used for performing a specific task are not ideal for it. Since the areas covered by these tasks differ widely from each other (and require different expertises), it would be too ambitious to study them all and solve all their issues at once. Moreover, the designing process, or the contents generation, of a video game has creativeness as core element, which makes it difficult to automatize processes such as the verification of correctness, or the quality, of a solution. In contrast, the programming process is more disciplined, since its core is based on logic, which can be automated and for which solutions can be formally verified. For this reason in this thesis we will focus on the programming process, by looking for the ideal formal/deductive mechanism (in the form of a language) to express game logic, which at the same time minimizes its development costs.

The motivation of our choice is also derived from the fact that the development of the logic of a game is a pervasive task that permeates the complete development process of a video game. During development, many versions are delivered before reaching the final version of the video game. Between these versions, the logic of the game might be subject to changes which happen more or less continuously. For example, designers might require small changes during the development process as response to some user testing, or might add a completely new game mechanic.

Due to its impact and importance, it makes sense to investigate the process of implementing the game logic, to understand its complexity and possibly find its limits and current issues. We believe that a scientific approach to solving some of these issues may benefit the whole process of developing a game.

1.3.2 Technological complexities

At this point we analyze the process of defining the logic of a video game. The logic of a video game is typically expressed by means of instructions that we give to the computer to execute. A computer interprets these instructions by means of some tools, which we can call software[1]. Software typically comes with a series of constraints (for example supported languages, allowed behaviors, etc.). A developer, while developing the logic of a video game, must always respect these considerations (for example if the software supports only a specific language then the instructions must be written in that specific language).

Since we cannot change the complexity of the intended game design, as it is imposed by the design of the video game itself, it makes sense to work on the complexity of the software. We now try to understand more about video games software complexity.

Software complexity

Software in and of itself is a complex structure, to the point that Frederick P. Brooks discusses software complexity as an essential property that cannot be ignored; in the following we quote this discussion, which is taken from the article entitled *No Silver Bullet: Essence and Accidents of Software Engineering*, of which we highlight in bold the steps significant for this section.

> "Digital computers are themselves more complex than most things people build: they have very large numbers of states. This makes conceiving, describing, and testing them hard. **Software systems have orders-of-magnitude more states than computers do**.
> Likewise, **a scaling-up of a software entity is not** merely **a repetition of the same elements in larger sizes**, it is necessarily an increase in the number of different elements. **In most cases, the elements interact with each other in** some **nonlinear fashion**, and **the complexity of the whole increases much more than linearly**.
> **The complexity of software is an essential property, not an accidental one**. Hence, descriptions of **a software entity that abstract away its complexity often abstract away its essence**. For three centuries, mathematics and the physical sciences made great strides by constructing simplified models of complex phenomena, deriving properties from the models, and verifying those properties by experiment. This paradigm worked because the complexities ignored in the models were not the essential properties of the phenomena."
> *No Silver Bullet: Essence and Accidents of Software Engineering by Frederick P. Brooks, Jr.*

Brooks also discusses that software complexity increases when the amount of interactions between entities increases. He also suggests a way to limit, or even solve, such issues and the proposed solution

[1] This is often, in the video game context, more than "just" a compiler of interpreter. See Chapter 2

is defined by two steps: (*i*) identify the essential properties of the domain touched by the problem, (*ii*) find how to express these properties within the software itself in the most natural way, in order to hide the complexities underneath them.

Video game software complexity

This discussion applies to video games a fortiori. Indeed, a video game is a high-level, real-time software application that allows the definition of entities and their interactions. Typically a video game features lots of interactions, where an interaction is made up of a piece of code that reads/writes the state of an entity. Since every entity of the game can be observed by many modules in the code, all running at the same time, a change in the state of one entity might potentially trigger such observers. If the state of an entity changes erroneously, for example, because of an error in logic, then the possibility to trigger observers that are not supposed to be activated is high. By accumulating these errors in the long term the state of the game will become unstable, therefore leading to unexpected behaviors in game. As a result, finding the origin of the error becomes difficult, especially if the error becomes visible to the developer after a series of concatenated chains of errors. Given the combinatorial complexity of game code with respect to the number of entities and their interactions, as a game grows in complexity, or dimension, the number of unexpected behaviors explodes as a consequence.

In addition, as games might also include non-functional properties, such as runtime performance, if the software used to develop the video game is not specifically designed to naturally capture such properties, then the complexity of related code will dramatically increase even more. This is due to the fact that these non-functional properties, which are necessary for the correct function of the game, must be integrated in the game code by hand by the developer.

For example, high-performance is a non-functional property that is common to most video games. In a video game like Asteroids shooter (a classic shooter game, where a player has to shoot and destroy as much asteroids as possible without being hit by them), the amounts of asteroids and projectiles might increase significantly during the game; thus the collision detection between asteroids and projectiles is a sensible aspect, which, if not treated properly, might significantly slow down the game performance. In a typical optimized solution, the game scene is divided into a grid, where every cell of the grid might contain projectiles, asteroids, or the player's ship. In the optimized solution, checking the collision between a projectile and the asteroids, requires the projectile to check only those asteroids in its cell, and those asteroids in the adjacent ones (in a naive implementation detecting the collision requires every projectile to check all the asteroids in the game). However, this optimization requires additional code and data structures, which we would not have in a trivial, but slow, non optimized implementation. This is due to the fact that the optimization described above is not mentioned in the original design of the game.

If this solution is implemented natively by the tool then the developer is only supposed to literally tell the tool what parts of the code require the collision detection, but if the solution is not supported natively by the tool then the developer is tasked to: implement the whole optimization, test it, and maintain it.

As these complexities increase in amount, risks, such as making mistakes, or unabeling to maintain the code, become realized problems. In this thesis we will focus on finding proper solutions to tackle such complexities and in particular, by focusing on those solutions that support developers with making less mistakes, and writing maintainable and readable code.

1.3.3 Video game developers

Making a video game (and software in general) is a time consuming activity. Moreover, game code needs to be maintained, upgraded, etc. and all these activities require time. Thus the longer a video game takes to be developed, maintained, etc. the higher will be its costs and effort to make.

We find three distinct kinds of video games developers: video games developers, serious games developers, and researchers. In the following we introduce these groups and discuss how costs affect their work:

- The category of entertainment game developers can be divided in two subcategories: game companies developers and independent developers. The reason of this division is due to the fact that these two distinct groups come with totally different budgets and typically different approaches in terms of marketing, game design, etc.
  - Game companies typically house a large number of developers. Their budgets, which are typically huge as well, can afford games that could take a few years to be developed. As making money is their main business element, when a design of a game becomes successful (in terms of sales) the later generations of games are often simple rearrangements.
  - Independent developers (also known as indie developers), typically feature small or medium-sized groups of developers. They are known for being innovative (their designs are often experimental), and are typically limited in terms of financial resources. Unfortunately, this limitation in terms of resources has an immediate effect on the quality and features of their games, so good products may lack important but hard to build aspects such as multiplayer, or advanced physics.
- Serious games developers typically are very small or medium-sized groups of people (sometimes even a single person) specialized in simulations for purposes other than entertainment, such as education, health care, city planning, etc. They are known for having limited resources [93], since their games do not enjoy the same sales as those meant purely for entertainment, although their social impact maybe be high nevertheless [75]. In their case costs have an immediate consequence for their productivity and the quality of their results.
- Researchers typically are small groups of people (sometimes even a single person) who use video games as frameworks for testing and simulating their research. It is often the case that researchers know little about video games development, because their expertise lies in other areas, such as medicine, social sciences, or engineering. Unfortunately, researchers do not enjoy big budgets in general, and typically can invest only small parts of their research budgets into the development of simulations. As the development phase increases in complexity the costs increase as well. As a result, the progression of their research is slowed down, as some scientific results are harder to obtain if the development of these simulations is slowed down due to the limited resources.

Since it is clear that costs for making games may be a great obstacle for making games, it makes sense now to investigate the tools used for making video games in order to understand how much these tools keep costs and complexity in check. In this thesis we will focus on finding proper solutions for making game to those developers with limited resources. Specifically, our target audience will be independent developers, serious games developers, and researchers

1.3.4 Current approaches

Modern tools for making video games, such as game engines (see Chapter 2.2.4), have an *abstraction mismatch* with their problem domain: they are either **too specific**, or **too general**. When **too specific**, a tool comes with a series of poor primitives that are effective just for limited genres of games (often at most one game genre is expressible). This limitation makes such tools not suitable for general game development. On the other end of the spectrum we find tools that are too general. When **too general**, a tool comes with a series of low-level primitives that require developers to specify a lot of details (often even unrelated to the game itself) to build a game.

1.4 Our focus and problem statement

For all the kinds of tools used for developing video games it seems that none of them provides a full solution to the problem of minimizing costs in game development without jeopardizing flexibility of use. This is a problem for all those categories of developers with limited resources, such as serious games developers, researchers, and indie developers. What is missing is a disciplined design that offers both the ease-of-use of highly specific tools, plus the openness of the general tools, all in one, i.e, a tool that minimizes the efforts for expressing games and their dynamics, while not being bound to specific genres. We look for such a tool in the field of programming languages and their abstractions, instead of purely focusing on engineering aspects such as libraries and frameworks.

Problem statement Our goal is to address the issues that arise from the above analysis. We can state the general problem statement as follows: **To what extent can a tool be built, which makes the complexities of general game development manageable for small and medium-sized teams of developers?** We argue that using specific tools and abstractions designed to naturally capture properties and elements of the domain of video games, rather than general purpose tools, would reduce the effort and costs of making games.

Research questions The research questions that we endeavor to answer in this thesis are:

1. What are the requirements that an ideal tool for game development needs to meet?
2. To what extent can a programming language for game development be built which meets the identified requirements?
3. How does such a programming language perform in terms of expressiveness, speed of execution, and maintainability, when compared to commonly-used tools for game development?

Positive consequences Games and simulations in general may have a substantial effect on our experiences. Virtual environments give us the opportunity to experiment with new ways to do research, to provide education, and to train and educate people. The fact that costs and efforts tend to rise considerably with the complexity of simulations, puts a lot of pressure on developers with limited resources. Our contribution to the state of the art is a reduction of the efforts and costs by supplying a computer language specifically aimed at game development. We believe that by empowering video games developers with effective and powerful abstractions (in our case in the shape of a programming language) for developing their games, which at the same time minimize the development costs, many positive consequences might follow, since the development time and costs are reduced. For example by avoiding to express details unrelated to the game logic, but necessary to the correct function of adopted tool, developers can now focus on exploring and researching innovative designs for connecting people, train students, and so on.

1.5 Contribution and thesis outline

In this thesis we explore the process of making the logic of a video game. We will study how video games are built by exploring the tools used in game development (Chapter 2). This study will lead us to insight into the advantages and disadvantages deriving from the usage of such tools. As we will see none of the tools used in game development offer more than a difficult trade-off between such advantages and disadvantages when tackling the complexity of game development.

We then present domain specific languages as a solution that achieves all the advantages deriving from the usage of the tools typically used for game development, while at the same time, avoiding many of the common disadvantages associated with such tools.

In particular we will present a domain specific language called Casanova 2 that is designed around the domain of video games, and the syntax and semantics of which is built ad-hoc to tackle the complexity of game development (Chapter 3). This novel DSL constitutes the first contribution of this thesis.

Casanova 2 comes with its own compiler, which transform the high-level, game-specific Casanova 2 code, into a high performance executable and without any specific intervention from the developers (Chapter 4). This novel compiler constitutes the second contribution of this thesis.

The semantic optimizations, which due to the specificity of the domain significantly improve the runtime performance of Casanova 2 code are further explored in detail (Chapter 5). This novel optimization constitutes the third contribution of this thesis.

Programming languages usually come with one or more implicit idioms for writing good programs. These idioms capture the essence of programming with the language. We will show the idioms of Casanova 2 and will use them to build actual games (Chapter 6).

We will evaluate Casanova 2 by means of two different types of evaluation (one qualitative, the other quantitative) to measure both the properties of the language and its attributes (Chapter 7).

Eventually, we will present the open challenges and future opportunities presented by this work, and draw our conclusions (Chapters 8).

In Appendix A we discuss the experience gathered from the first workshop on Casanova 2. In Appendix B we show how Casanova 2 works in practice by showing how it behaves when used by new developers who are not confident with it. In Appendix C we introduce the basic concepts of a multiplayer abstraction built in the Casanova 2 language supported with a concrete working example.

Chapter 2

Taxonomy of game development approaches

While it might seem desirable to encode games close to a high-level specification, the pragmatic reality has not, until very recently, allowed this. In this chapter we discuss the fundamental aspects that define a game and show how these aspects have been captured by means of game development tools. In particular, we begin with a formal mathematical introduction to what a game is and how its state changes according to the flow of time, and provide an example of a game structure (Section 2.1). We then discuss the issues arising from implementing such formalizations as a computer program. We present incremental solutions to these issues by relating each of them to a specific period of historical evolution in computer and programming languages (Section 2.2). Moreover, for each of these solutions we also discuss advantages and disadvantages deriving from their usages. Eventually, we propose a solution that solves all the identified disadvantages, and simultaneously covers all the advantages (Section 2.3).

Moreover, throughout this chapter we will support the discussion of each tool by means of one example, a moving particle. This example, which is on purpose small, in the beginning is explained formally, has the purpose of showing what kind of considerations and problems developers are faced with when designing and developing video games. We will see that the less a tool is suitable for developing games, the more details and effort it will require to build the example in question.

2.1 What is a game?

A game is any voluntary activity where people interact in order to achieve some goals within some constraints (described as *game rules*). The purpose of a game is to provide tools for the players that allow them to approximate their challenging expectations. These expectations may be provided externally, for example by an instructor, or may be self-motivated, like achieving entertainment[8].

2.1.1 Video Game

Within the panorama of games we find *video games*. A video game is a specialized kind of game where the interaction is carried out by means of electronic devices. Specifically, a video game, which from now on we will refer to simply as "game", is a *computer program* that continuously interacts with hardware components to carry out some game logic. The game automates the game rules mentioned above, therefore enforcing the structure of the experience[38]. Moreover, the program also handles

rendering, user input, the flow of time, etc. by providing a real-time experience that helps users to experience a "virtual reality" feeling[94].

2.1.2 Formal definition of a video game

In order to implement a game, we need a precise and formal detailed definition of its rules. Without such a definition we will not be able to "explain" to the machine what it is supposed to do. Therefore, in what follows we give a "pure" mathematical definition of a game that is technology independent and helps us to focus on the game definition only.
A game is made up of objects (each represented by a series of numbers), which we called *state*. In this formalization we can see a state $w(t)$ as a vector of all numbers that describe the game at some time t.

$$w(t) = C_1^t, C_2^t, \ldots, C_N^t \tag{2.1}$$

The dynamics of the game defines how the state changes over time. We can represent the evolution of the state by mean of an integrator that approximates each component of the state at all moments of the duration of a game[1] :

$$w(T) = \int_{t=0}^{t=T} \frac{\mathrm{d}w(t)}{\mathrm{d}t} dt \tag{2.2}$$

The integrator above computes the value for all components of the state. In what follows we see a *trivial* application of the above integrator to find the position of a particle, i.e, an entity with simply a position and a velocity, over time.

As an example consider a state $w(t)$ made up of a particle with velocity $v(t)$ and position $p(t)$:

$$w(t) = (p(t), v(t)) \tag{2.3}$$

According to (2.2), for this example computing the value of $w(t)$ requires first to solve the differential equation:

$$\frac{\mathrm{d}w_t}{\mathrm{d}t} = \left(\frac{\mathrm{d}p_t}{\mathrm{d}t}, \frac{\mathrm{d}v_t}{\mathrm{d}t}\right) \tag{2.4}$$

In this example, the velocity is defined as the rate of change of position with respect to time, and acceleration is defined as the rate of change of velocity with respect to time according to Earth gravity:

$$\frac{\mathrm{d}p_t}{\mathrm{d}t} = v(t) \qquad \frac{\mathrm{d}v_t}{\mathrm{d}t} = (0, -9.81, 0) \tag{2.5}$$

According to (2.5) at any time t, in this example, integrating the velocity $v(t)$ gives the position at time t, whereas integrating the gravitational acceleration returns the velocity at time t.

It might seem, at a first glance, from this example that solving the integral for each component of the state in isolation is sufficient to determine the value of the state. Unfortunately, this is usually not true. Typically games come with more complex dynamics: in a game the value of an object (the position for example) could be the result of combining different values of the state. For instance, the movement of a particle in a game may be subject to friction, or it may be influenced by collisions with other objects in the game world. Therefore, in most of the cases, since components of the state are tightly related to each other (with respect to time) the derivative of each component of the state depends on many elements of the state. For these cases the function to integrate is too complex and requires numerical methods to determine its values over time. In the following we discuss this issue and discuss the solution used for games.

[1] Components of the state might behave as discrete functions, for example a number that changes according to a timer. To treat such dynamics we treat their functions as *piecewise functions*.

Numerical vs. Analytic Solutions

The fact that we are able to model the evolution of the state by means of a function does not mean that finding an exact solution is possible or simple. This happens because the functions to integrate for the game will usually be too complex to allow analytical solutions: analytical solutions work only for simple models [81]. When the game becomes complex (imagine a city simulator, or a driving simulator with lots of physics) or the model is influenced by the user input, then it is not possible to identify a closed form solution [41].
We need to use numerical methods for solving game model equations such as the Euler method [11] (which is meant for solving systems of differential equations), where the initial values are the initial state and the update describes the changes of the state over a short amount of time.
We can use Euler to find the solution for the evolution of our particle. Consider (2.5):

$$\begin{aligned} \frac{\mathrm{d}\mathbf{p}(t)}{\mathrm{d}t} &= \lim_{dt\to 0} \frac{\mathbf{p}(t+dt)-\mathbf{p}(t)}{dt} = \mathbf{v}(t) \\ \frac{\mathrm{d}\mathbf{v}(t)}{\mathrm{d}t} &= \lim_{dt\to 0} \frac{\mathbf{v}(t+dt)-\mathbf{v}(t)}{dt} = (0,-9.81,0) \end{aligned} \tag{2.6}$$

By applying the Euler method to approximate the two limits in (2.6) we obtain the following:

$$\mathbf{p}(t+\Delta t) = \mathbf{p}(t) + \mathbf{v}(t) * \Delta t \qquad \mathbf{v}(t+\Delta t) = \mathbf{v}(t) + \mathbf{a}(t) * \Delta t \tag{2.7}$$

At this point, by taking many steps with small Δt (and an initial given value for time $t = 0$) for every component of the *State*, we achieved an approximated solution for the original integral shown in (2.2).
Of course this is an approximation. If we need higher precision methods then we could use better approximation methods such as those in the Runge-Kutta family[24].

Implementable formal specification

We now provide an algorithm that can effectively compute the state at any time t, given an initial state s_0.

With Euler we managed to describe how the dynamics of a game determine the evolution of the state for a very small amount of time. Unfortunately, if we try to increase such amount of time, then Euler alone is not enough[2][82]. Ideally, we wish to apply Euler, starting from an initial state, enough times until we reach a cumulative approximation of the state for the desired time.

For example, if we need the state for a time T, starting from an initial given state s, we apply once Euler to s for a small step dt and use the resulted evolved state for all successive applications of Euler. We keep repeating this operation until the amount of steps is enough to "cover" the whole desired time T.

We observe from the above example that Euler is used **at most once** per step. This is important for us, since we can now define a function *loop* that given a state s_0 and an amount of time t returns:

- s_0 in case t is less or equals to zero (which literally means what is the next evolution of s_0 for a step big 0);
- the application of a new state (obtained by evolving s_0 for a very small amount of time dt according to an Euler step) and an a decreased t (from which we remove exactly dt, the amount of time consumed by the Euler step) to a function ϕ. ϕ a high order function that unfolds a step of *loop*, by applying an Euler step once.

[2]Euler is a numerical approximation, small steps made of small amounts of time are necessary so to avoid to end into a wrong state.

$$loop_{\phi}(s,t) = \begin{cases} s, & \text{if } t \leq 0 \\ \phi(euler\_step(s,dt), t-dt), & \text{otherwise} \end{cases} \tag{2.8}$$

Of course, the above definition does not specify what happens after the single step of Euler: the ϕ function. To achieve the desired result, we need ϕ to continue with the very same process described by `loop` itself. This process, known as recursion, can be explicated by taking the *fixpoint* of the `loop` function[14]. The fixpoint operator will care to reapply `loop` to itself so that calling ϕ effectively calls `loop` again:

$$fix\ loop = loop_{(fix\ loop)} \tag{2.9}$$

In the following, we show how above formalism has been captured, since the beginning of the game development "era".

2.2 Game development

In the previous section we showed a symbolic representation of the dynamics of a moving particle and an equivalent numerical interpretation. Both descriptions are valid, although they differ in precision. The advantage of using the numerical approach is that we can implement it into a computer. Research in game development in the past decades was focused on finding suitable high-level interpretations for numerical solutions that work for all those "non-functional" pragmatic requirements such as real-time performance, networking, etc. (all these non-functional requirements add yet an additional challenge to research) [54]. A way to implement such high-level interpretations is by means of game making systems. Game making systems used to build actual games can be seen as ways to encode abstractions. The various historical game making systems, or game making tools, have always been intrinsically linked with the dominant programming languages and paradigms that were the most popular at the time of the tool in question. Each tool, with its language (and therefore paradigm [102]), imposed a set of limitations that ultimately were lifted by the next generation of tools [73].

This progression has clearly marched towards finally being able to write code against the mathematical specification and further away from hardware considerations.

Programming paradigms

Historically, as hardware has become increasingly powerful, programming paradigms less focused on hardware details have become usable in the practice of game development [79]. Among the possible paradigms used for making games we find: functional, declarative, object-oriented (OO), and procedural [68, 53, 40, 78]. By choosing a specific programming paradigm, game developers have to decide in advance how to design the architecture of the implementation for their games. These designs are shaped by the features offered by the chosen paradigm(s). For example procedural programming is "performance-oriented", OO programming is by some considered to be cognitively closer to the way humans perceive the real-world, declarative programming is meant for querying sets of facts and rules, and functional programming treats "all" computations as mathematical functions. Of course, every paradigm comes with disadvantages, which should be known in advance. For example procedural programming is not suitable for designing very complex architectures with clear separation of concerns, OO programming tends to add lots of overhead to the CPU, declarative programming is confined to query operations only, and functional programming programs are typically more complex to use, thus require a bit more of planning before writing the actual implementations.

The evolution of game making systems

Nowadays, we often see systems for making games that not only support programming paradigms for dealing with the game specification, but also provide sophisticated editors for building game content, kinematics, etc. [17, 61, 32, 63]. A chronological evolution of these systems for making games (inspired by [43]) is roughly reported in Table 2.1.

Table 2.1: Game making systems evolution

Period	Design philosophy	Languages	Paradigms	Discussed in
1950s	Hand made everything.	Assembly, C	Procedural	Section 2.2.1
1980s	Game making systems (no programming knowledge). User-derived, drag-and-drop visual interface engineered for the rapid prototyping of games.	_	Visual	Section 2.2.2
1980s	Graphic APIs. Developer oriented tools that provide a series of domain abstractions to deal with different hardware sharing similar functionalities.	HLSL, GLSL, PSSL, etc.	Declarative	Section 2.2.3
mid - 1990s	Low-level game engines. Developer oriented libraries that provide basic game functionalities in the shape of composable and reusable classes (such as physics, game loop, etc.) used inside game code.	C/C++, Java, C#, etc.	OO	Section 2.2.4
late - 1990s	High-level game engines. Typically come in the shape of tools that combine visual interface with actual coding. The visual interface is meant to deal with the common tasks of making games (assigning path finding properties to game entities, placing the game models on the map, defining the characters animations flow, etc.). Code is required to define special algorithms or game structures that are difficult to express with just the visual interface.	SGL, LUA, GML, Python, Casanova, etc.	Declarative, Functional, OO, Visual	Section 2.2.4

The fact that we classify a tool in earlier decades, does not mean that its philosophy is not used anymore. All the categories of tools and systems presented in Table 2.1 are still to some extent in use nowadays, with a tendency to use elements from older layers embedded into frameworks made according to the newest layer considerations. It is also worthy of notice that for many systems (independently of the level of abstraction) to achieve high performance multiple forms of ad-hoc optimizations have to be done by hand.

In what follows we discuss the elements of Table 2.1. In particular, for each element we discuss the pros and cons of using it, and show how one would implement the particle example presented in Section 2.1.2.

2.2.1 Assembly language (hand made everything)

Assembly language [20, 25] is the closest language to machine code. As for machine code, programs written in assembly can directly deal with CPU components such as registers. The goal of assembly

is to provide developers an abstraction over the binary format of machine code without losing the ability to directly manipulate hardware components such as the CPU or memory. This abstraction is achieved since assembly uses mnemonic operands to implement machine code.

Between the 1950s and the early 1990s most of the games were written in assembly code. The reason was that most of the games used to run on consoles, which used to come with limited hardware resources in terms of storage and computational power. Because of these limitations assembly language used to be the most suitable tool. Assembly instructions are limited in terms of CPU overhead and can produce high speed programs that work with limited storage space. Later, as hardware became more sophisticated and powerful, games started to feature higher level code such as C, confining assembly to the graphics and most performance sensitive code. For example, in Commander Keen the logic is written in C whereas code for drawing is written in assembly[3].

Successful examples Among the games written in assembly we find all those written for the Atari 2600, Apple II, Commodore 64, Atari 800, SEGA Genesis, the SNES, etc. In Figure 2.1 we provide some screenshots of games written in assembly.

Nowadays assembly is rarely used, since dealing with the low-level hardware components of the computer is achieved by means of standardized libraries. However, we occasionally find some traces of assembly code in libraries (although this is getting less and less common) from a few modern game engines.

Particle example - Assembly We now show how the "particle" example presented in Section 2.1.2 could be have been written in assembly[4]. Specifically, since the assembly code necessary to express the dynamics of our particle is not large, due to the intrinsic verbosity of the language, in Listing 2.1 we only present code for the position and velocity update.

Advantages The main feature of assembly is the ability to provide machine instructions to exactly specify all that the hardware must do, in extreme detail. The *absolute control* over hardware allows developers to write code with very high performance. This performance was crucial in games especially when consoles featured limited amounts of computational resources.

Disadvantages As the code above demonstrates, assembly is very verbose even to express a very simple operation such as updating the position of the particle. This is due to the fact that assembly does not provide effective abstractions for expressing high-level behaviors. By using assembly, developers are left the only choice of using low-level constructs that are tightly related to the hardware.

This limited choice pushes developers towards developing code that requires a lot of effort to be coded, as developers have to specify every single behavior of the hardware, including dealing with CPU registers or other hardware components. As a result of this, the chances of making mistakes are significant. Portability is also limited, since the choice of the assembly version is derived by the chosen hardware and different assembly versions come with different instruction sets.

Moreover, as CPU's become more powerful, bigger games and more complex low-level assembly instruction sets followed. This has slowly made it impossible to contain development costs without moving to more advanced tools with more sophisticated abstractions.

[3] https://github.com/keendreams/keen

[4] For this example we use the syntax of x86 assembly. The x86 assembly language differs from other assemblies, like MIPS assembly for example, and is meant for the class of x86 processors.

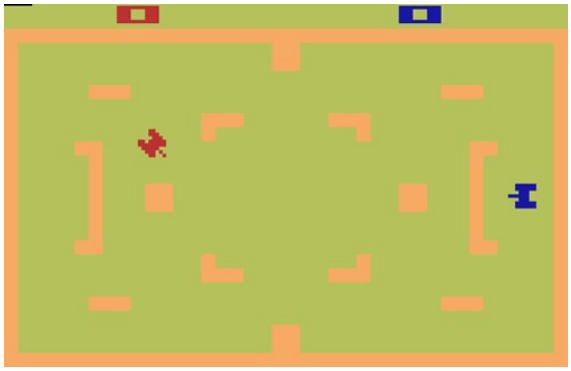
(a) Total carnage (1992)

(b) Commander Keen – Keen Dreams (1993)

(c) Prince of Persia (1989)

(d) RollerCoaster Tycoon (1999)

(e) NBA Jam (1993)



(f) Combat (1977)

Figure 2.1: Some assembly games

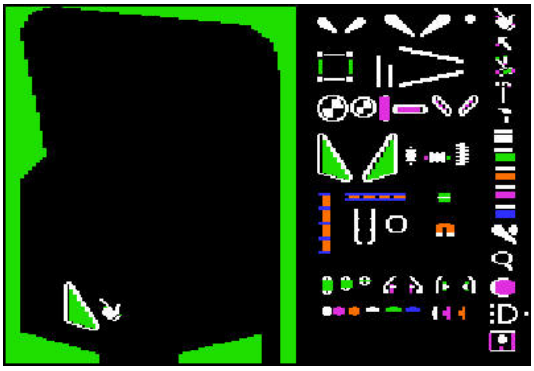
Listing 2.1: Particle velocity and position update in the Assembly lanugage

```
; 24   :      p = p + v * dt;
lea     eax, DWORD PTR _dt$[ebp]
push    eax
lea     ecx, DWORD PTR $T5[ebp]
push    ecx
lea     ecx, DWORD PTR _v$[ebp]
call    Vector2_times           ; Vector2::operator*
push    eax
lea     edx, DWORD PTR $T4[ebp]
push    edx
lea     ecx, DWORD PTR _p$[ebp]
call    Vector2_plus            ; Vector2::operator+
mov     ecx, DWORD PTR [eax]
mov     edx, DWORD PTR [eax+4]
mov     DWORD PTR _p$[ebp], ecx
mov     DWORD PTR _p$[ebp+4], edx
; 25   :      v = v + Vector2(0, -9.81f) * dt;
lea     eax, DWORD PTR _dt$[ebp]
push    eax
lea     ecx, DWORD PTR $T2[ebp]
push    ecx
push    ecx
movss   xmm0, DWORD PTR __real@c11cf5c3
movss   DWORD PTR [esp], xmm0
push    ecx
movss   xmm0, DWORD PTR __real@00000000
movss   DWORD PTR [esp], xmm0
lea     ecx, DWORD PTR $T3[ebp]
call    Vector2                 ; Vector2::Vector2
mov     ecx, eax
call    Vector2_times           ; Vector2::operator*
push    eax
lea     edx, DWORD PTR $T1[ebp]
push    edx
lea     ecx, DWORD PTR _v$[ebp]
call    Vector2_plus            ; Vector2::operator+
mov     ecx, DWORD PTR [eax]
mov     edx, DWORD PTR [eax+4]
mov     DWORD PTR _v$[ebp], ecx
mov     DWORD PTR _v$[ebp+4], edx
```

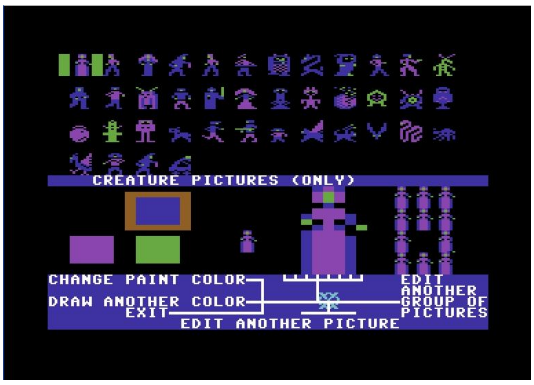
2.2.2 Game Creation Systems

A game creation system [33] is an expression tool designed around the domain of video games. The goal of a game creation system is to make game development accessible also to developers with no (or little) knowledge of computer programming, by simply allowing them to click buttons in a visual interface to define the entities of a game and their behaviors [26].

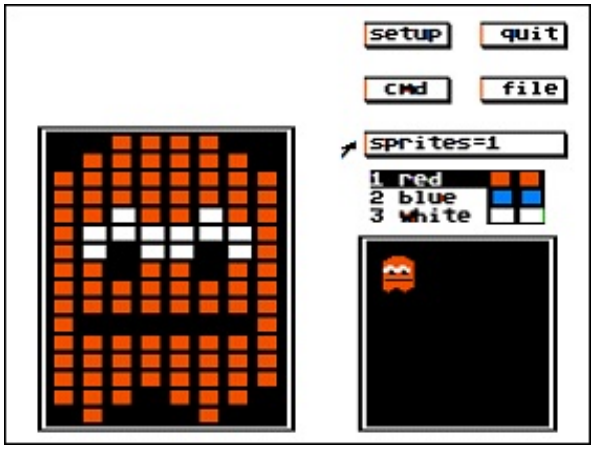
Game creation systems started to show off in the early '80s, when consoles and desktop stations started to became widespread. They were an exploratory parenthesis ahead of its time driven by the excessive low-level of alternative systems. ConstructionSet-Pinball, Garry Kitchen's GameMaker, and Adventure Construction Set (see Figure 2.2) are examples of such systems.



(a) ConstructionSet: Pinball (1983)



(b) Adventure Construction Set (1984)



(c) Garry Kitchen's GameMaker (1985)

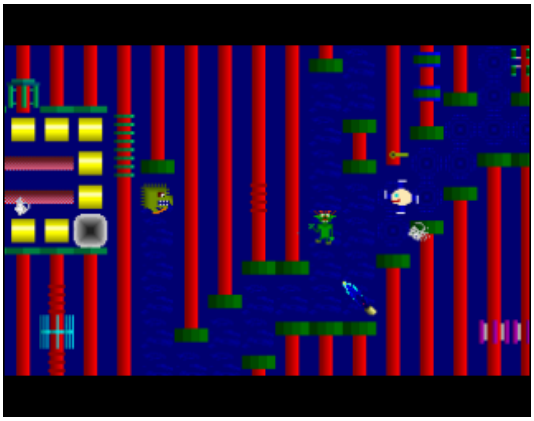
Figure 2.2: Some game creation systems

Typically, a game creation system focuses on a single genre of games plus a restricted set of similar subgenres. This is due to the fact that different genres share little logic. Therefore, expressing different game genres by means of just a visual interface (without the support of any programmable system) is difficult, if not impossible.

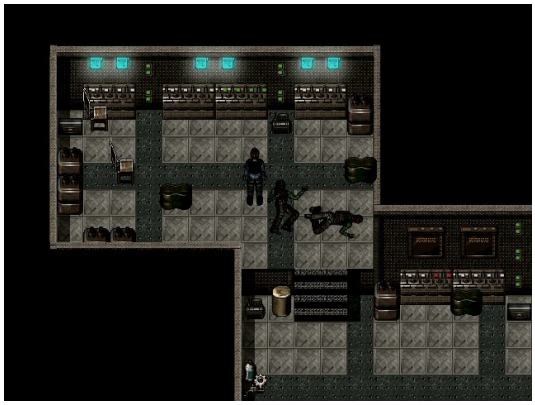
For adventure games we find: the inform language (1996), a text adventure language; Adventure Game Toolkit (1987), a program for adventure games development; RPG Maker (1995) and The

Bard's Tale Construction Set(1991), softwares for creating role-playing-games; The 3D Gamemaker (2001), a software that allows users to make 3D FPS's and adventure games; Game-Maker (1991) and Indie game maker (2014), general purpose software tools for game development. These tools are mainly used by small groups of developers, sometimes even by single developers.

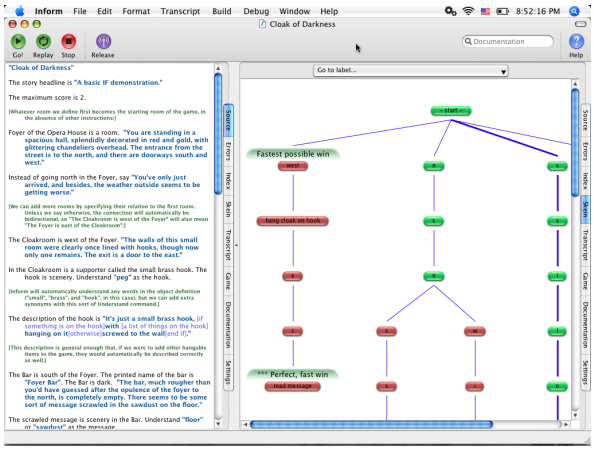
In Figure 2.3 we find some examples of games made with some game creation systems. Nowadays game creation systems are used less when compared to the past. Among the most active creation systems we find RPG Maker and Indie Game Maker editor.



(a) Pipemare (Game-Maker)



(b) City of Chains (RPG Maker)



(c) Slouching Towards Bedlam (Inform 6)

Figure 2.3: Games built with game creation systems

Advantages By targeting specific game genres, game creation systems can provide a domain interface that allows developers to effectively build a game with no knowledge of computer programming. The design of such visual interfaces is meant to allow developers to quickly prototype and test games, therefore reducing costs. Fast game prototyping, reduced game development costs, and domain interfaces are the main advantages of these tools.

Disadvantages Different games implementing different genres share little logic (for example, how much can we recycle from a solitary game into a shooter game?). In order to deal with such differences,

(a) Quake (1996)

(b) Wolfenstein 3D (1992)

Figure 2.4: Some software rendered games

different tools (each targeting specific genres) with ad-hoc interfaces were developed. Such differences made learning these game creation systems relatively expensive: whenever a developer changes genres he would have to learn another system. This task is not only time consuming but also is expensive in terms of effort.

The same issues apply to the customization of games made with such tools. As games got more and more sophisticated, the necessity for more powerful and expressive game creation systems piles up. Game creation systems try to tackle the expressiveness limitations of their visual interface, by extending/augmenting game creation systems with scripting facilities. Because of the lack of standardization of such scripting facilities and their poor integration in the system (game creation systems are not developed with scripting on mind), experienced developers prefer to choose more powerful and standardized tools such as a game engine.

2.2.3 Graphics API

A multimedia API (Application Programming Interface), such as OpenGL [87] or DirectX [67], is a set of routines, protocols, and tools. These API's were introduced in the early '90s for handling multimedia tasks (such as GUI, input, etc.) standardized across a variety of hardware platforms. Through appropriate abstractions developers could access the hardware of the computer, like the GPU, and make their code portable to different machines.

A graphics API is the best known example of a multimedia API centered around rendering tasks. The evolution of graphics API's on personal computers followed a very fast evolution curve that started in the '80s. Until the early '80s most of the graphics of games were written by manipulating the VGA (video graphics array) pixel by pixel in assembly or in C. By providing developers an array that represents the pixels of a monitor, developers could plot the desired colors into specific pixels (writing into that memory area would also write to the screen). Further evolutions allowed developers not only to deal with single pixels on the screen but also to draw textures, introducing the concept of 2.5D games, which featured 3D worlds rendered with no (or very limited) graphical hardware support. Among these software rendered 2.5D games we find Wolfenstein 3D and Quake (see Figure 2.4).

The CPU load of software rendered games was a known issue. As the necessity of high performance games and advanced 3D graphics started to become widespread among developers, modern GPU's came in to help with graphics acceleration. Thanks to graphics acceleration developers could finally delegate rendering tasks to the GPU while offloading the CPU. This made it possible to achieve higher performance, since the GPU is designed to process graphics commands in parallel and has dedicated

memory. Moreover, this would free the CPU to process game logic such as AI, physics, networking, and other tasks, thereby significantly improving the overall game experience.

Graphics API's for accelerated hardware In the early '90s, because of the increasing complexity of GPU's, a new generation of graphics API's (Application Programming Interface) was introduced. The goal of such API's was to abstract the complex hardware of modern accelerated GPU's in favour of a high-level model of its behavior. Such a model would help developers with expressing graphics directives with little effort, and help developers stay focused on the design of graphics effects algorithms rather than having to think constantly about specific hardware details.

Among such API's we find IRISGLP, OpenGL (an improved version of IRISGLP), Glide, and DirectX. Nowadays DirectX and OpenGL are the most used graphics API for rendering game contents.

FFP (fixed function pipeline) Abstracting the complex hardware of GPU's became possible due to the introduction of the so-called FFP (fixed function pipeline). Fixed functions are a series of functions that map directly to dedicated drawing logic that can only be used on GPU's designed to support them. By editing a set of hardware switches, developers could customize those functions. However, this customization comes with some expressiveness limitations, since editing the hardware switches allows developers to customize single or small groups of instructions but not the fundamental shape of the underlying algorithms. Moreover, since the hardware switches are shared among several functions, making predictions on the algorithms behaviors became complex and hard: changing just one switch might affect the behavior of the FFP dramatically.

These limitations pushed the community towards the development of a better abstraction mechanism that would lift the artificial limitations of the FFP.

Shaders To overcome the FFP limitations, customizable pipelines were made programmable through the system known as "shaders", or "programmable pipelines". By introducing shaders, which are small programs that are run on the GPU pipline, developers could design their own algorithms and have a clear control over the pipeline process.

With shaders, a developer could manipulate the pipeline in two different processing stages: vertex processing and pixel processing (Figure 2.5). For vertex processing, the developer has the task of designing an algorithm for placing every game element from model space to world space. For pixel processing, the developer has the task of designing an algorithm to draw the game elements that are inside the frustum of the camera to the screen (pixel by pixel). New shader models have more stages that are programmable.

Fixed functions vs Shaders Fixed functions represent the first attempt to make customizable GPU pipelines by providing developers with a series of functions that can be customized to specific drawing scenarios. Shader systems are programmable instead and allow developers to deal with graphics data (or game geometries) by means of user-defined algorithms that define how those graphics data are transformed and rendered.

Particle example - FFP/OpenGL/C++ In Listing 2.2 we show a complete solution to the "particle" example presented in Section 2.1.2 written in C++ and using OpenGL as graphics API with the FFP .

Particle example - Shader/OpenGL/C++ To benefit from shaders, the previous code requires some adjustments. First we need to define our vertex and fragment shader. In this example we wish to change the color of our particle to green and to make it smaller.

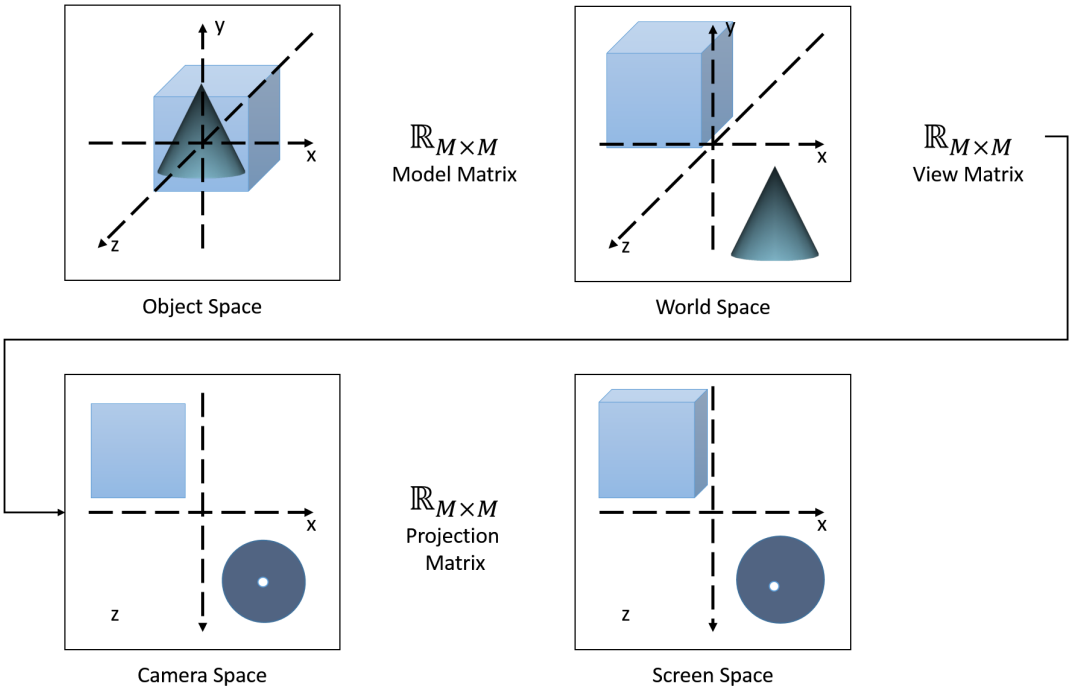


Figure 2.5: Drawing stages of modern GPU's

Listing 2.2: Particle written with C++/OpenGL

```
#include<GL/glut.h>
#include "math.h"

Vector2 position = Vector2(0, 0), velocity = Vector2(0, 0.0001);
void glutInitRendering() {
  glEnable(GL_DEPTH_TEST);
}
void reshaped(int w, int h) {
  glViewport(0, 0, w, h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(45, 0, 1, 200);
}
void update() {
  position = position + velocity;
}
void display() {
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glClearColor(0, 0, 1, 0);
  glPushMatrix();
  glColor3f(0, 1, 1);
  glTranslatef(position.x, position.y, 0);
  glutSolidSphere(0.1, 23, 23);
  glPopMatrix();
  update();
  glutSwapBuffers();
}
int main(int argc, char **argv) {
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
  glutInitWindowSize(400, 500);
  glutCreateWindow("Bouncing Ball");
  glutInitRendering();
  glutDisplayFunc(display);
  glutIdleFunc(display);
  glutReshapeFunc(reshaped);
  glutMainLoop();
}
```

- The following vertex shader scales all vertices in `x` and `y` direction.

Listing 2.3: A simple vertex shader

```
void main(void)
{
  vec4 a = gl_Vertex;
  a.x = a.x * 0.5;
  a.y = a.y * 0.5;
  gl_Position = gl_ModelViewProjectionMatrix * a;
}
```

- The following fragment shader sets to green the color of all pixels corresponding to the particle on the screen.

Listing 2.4: A simple fragment shader

```
void main (void)
{
  gl_FragColor = vec4(0.0, 1.0, 0.0, 1.0);
}
```

Once the shaders are defined, we need to load them into a shader object within the `glutInitRendering` function.

Listing 2.5: Loading the shaders

```
void glutInitRendering() {
  glEnable(GL_DEPTH_TEST);
  //SM is properly initialized variable of type glShaderManager
  shader = SM.loadfromFile("vertexshader.vs", "fragmentshader.ps");
}
```

To use the shader object, we need (inside the `display` function) to call in order the methods "begin" and "end" of the shader object and to put the actual drawing calls within these two calls (Listing 2.6).

Advantages API's set a new stage in game development, by providing developers an easier abstraction experience compared to coding everything in assembly. By means of a shader, for example, customizing the behavior of the GPU becomes more accessible. Moreover, hardware considerations are, to some extent, hidden to developers. Indeed, developers are not required to master memory, CPU vector instructions, etc. to achieve high performance, since every operation in a shader maps to complex hardware instructions.

Disadvantages API's provide generic abstractions for game development that add a level of complexity to the task of making games. A developer, in order to make a game, is now also tasked with understanding and mastering the chosen API, which for many cases comes with its own domain specific languages for various internal tasks. Moreover, in many cases, developers are also asked to learn and master other domains, like math, to effectively use the selected API, thus adding yet another layer of complexity. For example, when dealing with shaders, math is important in order to apply any form of visual effects, from basic linear algebra in vertex transformations to approximation of complex integrals for lighting computations.

Listing 2.6: Calling the shader

```
void display() {
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glPushMatrix();
  glTranslatef(position.x, position.y, 0);

  shader->begin();
  glutSolidSphere(1.0,32,32);
  shader->end();

  glPopMatrix();
  update();
  glutSwapBuffers();
}
```

These layers of complexity make the learning curve of such API's steep, further affecting the costs of game development.

2.2.4 Game Engines

A game engine [19] is a tool designed to abstract the development process of a game and is used to develop games for different platforms such as consoles, desktop PCs, mobile phones, etc.

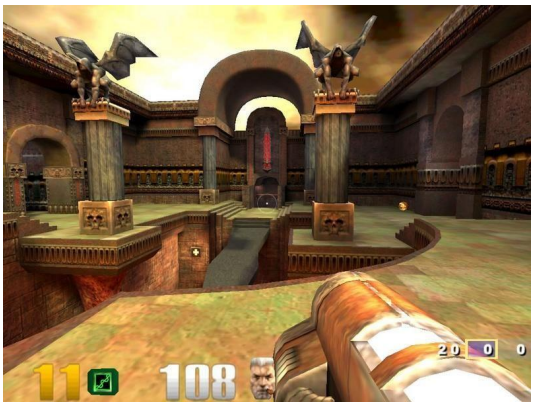
The goal of game engines is to provide a series of reusable abstractions that can be composed in order to provide an effective extension tool that allows the tackling of a variety of different scenarios. This composition property, coupled with a relatively low number of abstractions available, means that we can now effectively face a variety of problems in game design without having to resort to building everything from the ground up, or use a broad variety of different tools, one per specific scenario.

Moreover, with the rise of 3D games and the increasing computational power of hardware in the '90s, the complexity of games increased. Games started to implement features such as sophisticated artificial intelligence, complex rendering effects, and networking, to satisfy consumers' needs that added yet a further layer of complexity to the task of developing video games. Since these complex features were difficult to implement with traditional tools (due to their limited abstraction capabilities) the necessity for more expressive tools with higher abstraction, and specifically targeted to the domain of games mechanisms, became relevant. For this purpose game engines were developed. Typically a game engine provides several components, each of which is designed for dealing with specific game development tasks such as physics, levels editing, rendering, sound, AI, networking, localization, input, etc.

Game engines became very popular in the mid-1990s after the ground breaking titles Doom and Quake made their appearance. The success of Doom and Quake was so dramatic that other developers and companies wanted to reuse elements of such games for their titles, so Id Software (and later Epic Games's with the Unreal series) designed successive versions of their game codes with reuse and extensibility in mind: first the game engine is implemented (made of composable and programmable modules) then the game engine is used to implement the game. Some of the best known game engines are OGRE, XNA, Blender, Unreal Engine, IdTech, and Source. In Figure 2.6 a series of games built with some of these game engines are shown.

Game engines differ from each other based on the level of detail provided to developers to control and customize [9]. We group such engines in the following two categories:

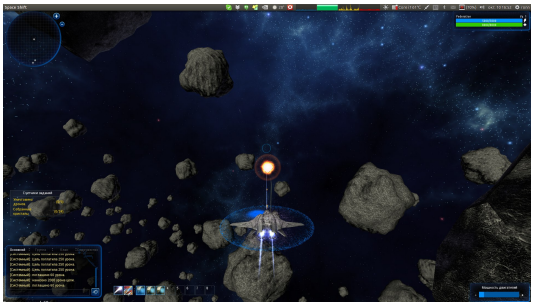
(a) Star Wars Jedi Knight: Jedi Academy (idTech3)



(b) Quake III Arena (idTech3)

(c) Torchlight II (OGRE)

(d) Magicka (XNA)



(e) Space Shift(jMonkeyEngine)

(f) Half-Life 2 (Source)

Figure 2.6: Games built with custom engines

- **Low-level engines** are engines where a series of libraries are provided within some frameworks. These frameworks typically provide developers with basic games abstractions such as the game loop, contents loading facilities, etc. Typically, low-level game engines give the most flexibility and performance, but they are expensive to build and use, since the developer has to maintain, program, and connect each component used in the engine. Customization is possible by means of fully fledged, general purpose programming languages that are used not only to connect the components but also to define the logic of the game. XNA [80], Pygame [98], jMonkeyEngine [64], Löve [6], etc. belong to this group of engines.

- **High-level engines** are sophisticated game engines that come ready out of the box. The goal of such engines is to reduce the complexity of developing games by providing already made components that do not need developers to adapt them or connect them, since they are already connected and integrated in the engine. Developers are only tasked to use such components and compose them, typically by means of a GUI, to build their games. Customization in high-level engines is possible, but by mean of general purpose languages (GPL's). OGRE [95], Unreal Engine [44], Torque Game Engine [66], id Tech [90], etc. belong to this group of engines.

Particle example - XNA/C# We now show a complete solution to the "particle" example presented in Section 2.1.2 written in a low-level game engine. Specifically, we use for this sample XNA as game engine.

Listing 2.7: Particle written in XNA and C#

```
public class Particle
{
  Vector2 particle_position,
          particle_velocity = Vector2.One * 100;
  Texture2D texture;
  public Particle(Texture2D texture)
    { this.texture = texture; }
  public void Update(float dt)
    { particle_position = particle_position +
                          particle_velocity * dt; }
  public void Draw(SpriteBatch sprite)
    { sprite.Draw(texture, particle_position, Color.White); }

}

public class MyGame : Game {
  ...
  Particle particle;
  protected override void LoadContent()
  {
    spriteBatch = new SpriteBatch(GraphicsDevice);
    particle = new Particle(Content.Load<Texture2D>("circ.png"));
  }

  protected override void Update(GameTime gameTime)
  {
    //Logic code goes here...
```

```
    particle.Update((float)gameTime.ElapsedGameTime.TotalSeconds);
    base.Update(gameTime);
  }
  protected override void Draw(GameTime gameTime)
  {
    //Drawing code goes here
    GraphicsDevice.Clear(Color.CornflowerBlue);
    spriteBatch.Begin();
    particle.Draw(spriteBatch);
    spriteBatch.End();
    base.Draw(gameTime);
  }
}
```

Advantages Game engines are a great result of applying software engineering techniques to game development, such as composability and resuability, for the definition of a series of abstractions all meant to reduce costs and to support developers into the definition of games and on a variety of game design problems like defining advanced AI, networking, content, etc.

Disadvantages Despite their power, engines suffer from severe limitations. These limitations are different depending on the architecture of the engine: low-level and high-level.

Low-level engines lack game-specific facilities to create a game due to the fact that these engines only offer libraries meant for general usage (libraries are unaware of the specific context in which they are going to be used). Building games then requires writing large amounts of complex game specific code, such as a path finder, optimizations, AI, etc. implemented by means of GPL's.

High-level engines provide a large amount of existing components that are a potential fit for many games. Unfortunately, being able to effectively choose and use a high-level engine requires developers to read large volumes of documentation.

Games that do not fit the standard components implementation can still be implemented, but this requires customizing components. Typically, customizing such components is done by means of GPL's and requires large amount of complex game specific code.

2.3 Discussion

In the previous sections we analyzed tools for making games and discussed their features. Moreover, for every tool we discussed advantages and disadvantages of its usage. In the following, we provide a summary of these features. Every item of the table is the result of grouping common features among the different, previously presented, tools. Indeed, for every item we also indicate what tools are affected by it.

In the following we discuss every disadvantage and advantage introduced in Table 2.2.

Disadvantages

As highlighted by the variety of tools, no solution has so far proven to be definitive. Nowadays, we see a variety of game development tools (Unity3D, Unreal Engine, MonoGame, etc.) each specialized, or simply working better, on specific areas of the game development panorama. This variety of tools is also motivated by the fact that many of these tools are little more than "major rearrangements" of previous tools (see for example the Unreal Engine series). Each of these rearrangements, which

Code	Disadvantage	Involved tools
D1	Verbosity	Assembly
D2	Portability	Assembly
D3	Learning curve	API, Game engines
D4	Specificity	Game creation systems
D5	Performance	Game creation systems
D6	Gluing	Game engines

Code	Advantage	Involved tools
A1	Writing	Game creation systems
A2	Reading	Game creation systems, API
A3	Optimization	Assembly, Game engines
A4	Interoperability	Game engines
A5	Genericity	Game engines

Table 2.2: Disadvantages and advantages of tools for game development

in many cases is simply the result of engineering trade-offs, has the task of fixing or compensating limitations of the previous generation. However, the lack of a disciplined, scientific approach has led to some structural, recurring issues, which plague multiple systems across different generations. In the following we describe such issues, presented in Table 2.2.

Verbosity: the lack of high-level abstraction results into more effort needed to express certain aspects of a game. This translates into additional costs to develop a game, as for example more code yields more errors in logic or runtime errors, less maintainable code, etc. These costs becomes even more severe when a tool is not meant specifically for game development, but is rather meant for generic domains. For example, with a general purpose language (GPL) expressing game domain specific behaviors, such as those depending on the flow of time, will typically require developers to write complex, possibly nested, state machines. This is a time-consuming, error-prone task.

Lack of portability: a tool that requires developers to include in their solutions aspects that depend on the adopted hardware, or on some very tool-specific features, for example, in Unity3D entities in a scene are accessible through dictionaries, will make such solution dependent on the given tool. This translates into additional costs when supporting a variety of systems, as different solutions become necessary to address different platforms given the same game logic.

Steepness of learning curve: a tool for game development that requires developers to include in their thinking process some considerations, which are not directly related to the game itself, but rather to the specific tool idioms, will require additional time to develop a game. This translates into additional costs when developing a game.

Lack of customization: some tools come with specific interfaces that allow the definition of limited game genres (sometimes even one genre per tool). This lack of customization makes tools bound to specific genres and add constraints to the design space. As games and their genres evolve in complexity more tools become necessary to express these changes. Moreover, mixing different genres becomes a challenging issue.

Low performance: some tools for game development are meant to express any sort of game, thus using generic and composable containers and components (with basic functionalities) to express game entities and their behaviors. Typically, such components are built in the tool, and instantiated by the game scripts. When a tool starts featuring many of these components then run-time performance is

affected negatively, due to the impact on cache coherency, virtual tables, etc. caused by abstraction mechanisms of modern (OO) languages.

Gluing frameworks and libraries : as games evolve in complexity, tools which lack customization facilities become less suitable to express such games. Thus, in order to deal with this issue, tools end up allowing third party tools or libraries to interface with them. These third party tools or libraries, which could be for example a scripting language, might not be aware of the tool mechanics or lack smooth integration. Thus developers are supposed to understand the mechanics of both the external tool and the main tool for developing the game, in order to effectively use them, plus a layer of "gluing" facilities which are often complex in themselves. This ultimately adds additional costs (complexity along common boundaries between the different tools, etc.) to the task of making a game.

Advantages

The booming success of tools is motivated by some clear advantages that can be reaped. For many tools these advantages are the reason of their success, as these advantages reduce the costs of developing a game. We present these advantage below (they have already been introduced in Table 2.2). The following advantages are scattered among all game development tools (and never available all at once in a tool). These advantages should be taken as a source of inspiration for future generations of game development tools.

Writing: a tool designed around the domain of games typically comes with some features designed specifically around the definition of some game aspects. The goal of these features is to speed up the process of expressing those game aspects by means of appropriate abstractions. These abstractions are chosen such that less constructs will be needed to express, for example, a complex decision tree, spatial indexes, etc. As a result, developing a game becomes less expensive, as less time and effort will be required.

Reading: a tool designed around the domain of games typically comes with features to capture some specific game aspects. These features positively affect verbosity, as less idiomatic elements "not-related" to the game itself become necessary. This in return improves maintainability and readability, as less idiomatic elements are necessary to express games logic. As a result, developing a game becomes less expensive, since less time and effort are required.

Optimization: a tool designed around the domain of games typically comes with some features designed around the *runtime* behavior of some game aspects. It is often the case that these features are translated automatically by the tool into equivalent, but more efficient, executable artifacts. For example, a tool might optimize some queries[5] in a game by adopting spatial indexes without the direct developer intervention. As a result, costs are reduced as these typically complex optimizations are streamlined, and code retains high performance and remains well readable.

Interoperability: a tool that allows interoperability with third-party tools, typically comes with a series of boundaries or requires adapters. These boundaries or adapters are meant to maintain the original tool identity even when interfacing it with third-party tools that come with different philosophies. As a result, costs are kept in check, since developers can now focus only on the chosen

[5]"Queries", throughout this work, refers only to in-memory endomorphisms in the domain of collections of entities and should not be confused with SQL-style database queries. This slight misuse of the term queries comes from the fact that game engines share plenty of architectural considerations from DBMS's

tool, and its philosophy, instead of breaking down the program into pieces, where each piece is designed and implemented to include considerations of the different external tools.

Genericity: a tool designed around the domain of general game development typically comes with a series of building blocks that are designed generic enough to support a variety of specific games. To build a game, developers are thus only tasked to properly combine such blocks to achieve the desired result. Since the amount of building blocks to master is limited and since they tend to be broadly applicable, developers can focus on the logic of the game and its design, and how to express it to the lby always using the same set of primitives instead of a separate tool (and its accompanying building blocks) per game or per genre. Therefore, the genericity of a general programming language for video game development yields less cognitive stress due to a reduction in the number of building blocks available. Therefore, by means of generic tools for general game development, developers can focus on the core of the game development process which then becomes more efficient, and thus also less expensive.

2.4 The necessity for a domain specific language

So far, specific problems in games have been tackled with more and more domain specific tools (DST's) such as Unity. The limits of such tools were made less dire with extensibility, usually by means of a general purpose language (GPL). However these GPL's lack the domain specific abstractions of games, leading therefore to highly complex code that is expensive to maintain and develop. Indeed, modern GPL's are particularly weak when dealing with properties typical of the domain of games such as: concurrency over shared resources among game entities, distributed code in networked games, efficient event handling, and time manipulation.

In order to continue our search for better abstractions it makes sense that we now focus on GPL's in order to augment our domain specific tools with domain specific languages (DSL's) [74]. A DSL is a specialized language [42], typically small and very expressive, aimed at solving only problems within the chosen domain through an optimal choice of operators, abstractions, and level of focus. Attention on DSL's has increased in recent years, since mapping all the requirements of games with game tools exclusively is difficult and expensive (this difficulty gets even higher when variations in the requirements occur often, requiring to break the careful mixture of GPL code and tool settings found so far). Research in game development is pushing nowadays towards the study of such DSL's in order to provide additional support over different game tools. By means of such DSL's, game tools (and game development in general) would benefit from the advantages defined above:

- Writing: a DSL provides domain abstractions that can, for example, speed-up the developing process of a game and so reducing costs. This is also due to the fact that complex behaviors or interactions in a game can be expressed in code with few domain-specific constructs.

- Reading: since the abstractions provided by a DSL are built ad-hoc around a domain (in our case the one of video games), reading and maintaining code written with it becomes more intuitive. Moreover, ad-hoc abstractions make code more compact; this positively affects readability as less words are necessary to understand complex behaviors, leading to more efficient use of the reader short-term memory[12].

- Optimizations: since a DSL provides abstractions that capture complex domain behaviors or interactions, these abstractions can be used by the tool supporting the DSL (typically a compiler) to refactor the code in order to achieve better performance. As an example see how a DBMS optimizes SQL code (in this case SQL is the DSL and the DBMS is the supporting tool).

- Interoperability: a DSL is small and built ad-hoc to react to variations of the domain in question. Thus, when a third party tool or library needs to interact with the game code, there are two possibilities that preserve the DSL nature and allow such interoperability: (*i*) by means of new constructs that capture the fundamental aspects of the third-party tool or library (these constructs will make sense only in the context of the DSL in question), or (*ii*) by a layer that is built at the compiler level and that acts as an adapter between the DSL and the third-party tool or library (in this case the compiler should provide some clear and easy-to-use interface).

- Genericity: a DSL provides a series of generic building blocks designed around a specific domain. These blocks are generic enough to support the variety of programs of the domain in question. Since the number of blocks is limited, developers can focus better on the logic of the program, instead of requiring them to learn different tools to tackle needs of different programs. This results into less cognitive stress, which eventually leads to more productivity.

Of course by achieving the above advantages, the disadvantages fade out, as they are (to some extent) complementary to the above advantages. For this reason we do not discuss them further in this section.

It turns out from this analysis that we need a language to achieve the advantages and solve the disadvantages presented in Table 2.2. In particular, it turns out that a DSL seems to be a valid solution. In what follows, we present our solution to this problem. More in detail, we present a concrete DSL (Casanova 2) and show how our DSL incorporates all the advantages while avoiding the disadvantages listed in Table 2.2.

Chapter 3

The Casanova 2 language

In this chapter we present our solution to the problem of creating games, using a tool that avoids the disadvantages listed in Table 2.2, without sacrificing the advantages listed in the same table.

This solution comes as a series of *proper abstractions built ad-hoc around the domain of games.* These abstractions come in the shape of general building blocks that can be composed into infinite shapes, with structural correctness as the only limitation. The reason for this limitation is the fact that structural correctness helps developers to avoid errors, such as runtime errors, logical errors, or compilation errors, by enforcing only compositions between building blocks that are reasonable for the domain. In the following, we will provide an implementation to these abstractions, which comes in the shape of a Domain Specific Language (DSL) called Casanova 2. The goal of Casanova 2 is to help game developers in reaching their goals by substantially reducing development efforts, with a special benefit for small and medium-seized game development teams.

We begin with a discussion to identifying the complexity of games code by introducing a case study. We use this case study to identify issues in the way games are traditionally expressed (Section 3.1). We then introduce Casanova 2 as a tiny, concurrency-oriented, game-centered language for describing game logic, and show how the case study is expressed in this language (Section 3.2). We round off with the conclusions for this chapter (Sections 3.3).

3.1 Technical challenges in games development

In this section we discuss games and their complexity by means of a case study. We consider an example showing the complex interactions that are typical for games, in the form of the state of the game and its continuous and discrete dynamics.

3.1.1 Running example in pseudo-language

The running example we use is a patrol moving through checkpoints. This example features two sorts of game dynamics in a minimal way: (*i*) continuous when the velocity is applied to the position of the patrol at every game iteration; and (*ii*) discrete when the patrol chooses the next checkpoint after he reaches the current one. The state of the patrol is made up by the position of the patrol `P`, and its velocity `V`.

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

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

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

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

Dynamics such as the one described above are built in games either with engines or by hand. Game engines often provide already made components for typical game dynamics. However, game engines are often difficult to expand or customize, hence specific behaviors, such as the one described above, will require developers to implement them by hand (possibly via a programming language). Thus, we will now focus on the scenario when such dynamics need to be built by hand.

Hand made implementations A hand made implementation is used when developers (who are looking for specific behaviors): (*i*) want to have more control over the game implementation, (*ii*)face the problem that the support of the underlying platform is poor, or (*iii*) want to build anything that is not readily supported by existing libraries or engines.

Hand made implementations raise important issues to be considered before starting a new project since:

- Games tend to be very large applications. As size increases, the number of interactions between game entities, or code modules, increases as well, together with the risk to make mistakes; and
- Hand made optimization adds complexity, because it requires supplementary data structures and may subtly affect the actual game logic. Optimization may also lead to (*i*) implementation issues (for instance some optimization may work only on specific architectures), and (*ii*) maintainability issues (any change in the game design should keep into account its effects on the implementation).

We now present an example of a hand-made implementation of the patrolling dynamics following the style of [76]:

```
class Patrol:
  enum State:
   MOVING
   STOP

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

  def loop(dt):
   P = P + V * dt
   if myState == MOVING:
     if P == Checkpoints[currentCheckpoint]:
       myState = STOP
       V = Vector2.Zero
       timeLeft = 10
```

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

The `loop` function implements the patrolling behavior. It takes one argument, a `dt`, which represents the delta time elapsed since the last frame.[1]

The very first line of the `loop` body implements the position update behavior. The velocity behavior depends on whether the patrol is moving or idle. While moving, we stop the patrol as soon as he reaches the checkpoint, and set the wait timer to 10 seconds. If the patrol is idle and the countdown is elapsed, the next checkpoint is selected. At this point the patrol points toward the new checkpoint and starts moving again.

3.1.2 Discussion

The patrolling sample illustrates what is often a semantic schism between design and implementation in games. Deceptively simple problem descriptions turn out to require surprisingly articulated implementations. Complexity mainly originates from the explicit definition and management of a series of *spurious variables* that are needed to program the logical flow of the problem but which do not come up in the design. In our case study, which is trivial, we already have spurious variables: `myState` (together with the definition of the state structure) and `timeLeft`. Moreover, the if/elif structure, the lookup in the array of checkpoints, and the % operator to avoid out-of-bounds errors, represent yet more noise in the code, further obfuscating its meaning.

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

3.2 Casanova 2

Languages, in general, offer more expressive power than engines, because of their ability to combine and nest constructs. An **engine** typically can be thought of an already made machine that comes with a series of "on-off" switches and parameters. In order to obtain the desired behavior a developer has to interact with these switches by turning them on and off accordingly. However, expressiveness of these engines and customization are limited, as the amount of possible states is limited by the amount of available switches. A **language** is much more expressive than an engine, since it features a tree of switches, with mutually recursive references: recursiveness allows the definition of potentially infinite amounts of combinations of these switches.

A language specifically designed and built with game programming in mind can help with common aspects of game development (such as time, concurrency, and state updates) that regular languages

[1] The delta time of a game can be either **fixed** or **variable**. When **fixed**, each update moves all entities by a fixed time period. This can facilitate some tasks such such as the one of debugging and testing, since every movement is deterministic and thus predictable. When **variable**, each update moves all entities by an amount of time that is the time difference between the last update and the current one. This gives the game an indication on how long it took to process and draw the game state, and therefore how much the various entities should "move" on screen for the user to perceive no mismatch between real-time and game-time.

do not encompass. In this regard, we present the language Casanova 2, based on [70], which takes its inspiration from the orchestration model of [77]. We show how Casanova 2 is designed in particular to express the typical dynamics present in games.

3.2.1 The basic idea behind Casanova 2

An abstraction of a game should be able to represent its main elements, i.e., its state variables and their (discrete and dynamic) interactions and nothing else (thus no noise). For this purpose, we built an (intentionally) small programming language of which the main features are *state* and *rules*:

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

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

3.2.2 The running example in Casanova 2

We can now show how to rewrite the patrol program presented in Section 3.1 using Casanova 2.

Listing 3.1: Patrol in Casanova 2

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

   rule P = P + V * dt

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

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

In Casanova 2, as shown in patrol example, a fundamental design aspect is the interruptibility of any block of code through specific constructs. The `wait` construct interrupts for a given amount of time, whereas the `yield` construct interrupts but also updates the value of the fields declared on top of the rule. The `yield` construct is the only way to produce an *observable* side effect. This means that in order to transform the state of a game developers can only use the `yield` construct. Moreover, any construct, such as `if`, `for`, or `while`, can be nested or combined with the `yield` and `wait` constructs. Therefore, all such constructs can represent "atomic" operations, which can be executed all at once, but also "real-time" operations, which take a (purposefully) much longer time to run to completion. This matches the human intuition of terms such as "while", for example in the sentence "while these are enemies nearby, do stay under cover". Such a mechanism would not be implementable with a "while" loop in a traditional programming language, but would rather become a cascading "if-then-else" or a "switch" as in the example above.

Another crucial aspect lies in the controlled propagation of side effects as a result of the execution of rules. Each rule must exactly define which fields (of the entity it runs from) it will potentially change. The rule may only *read* the other fields, and also the fields of the root entity and child entities. As a consequence, communication between entities follows a strong, predictable, hierarchical discipline. Compared to raising events, or even directly writing the fields of child entities, this enforced discipline prevents all the problems such as "callback hell", cyclic events, or undesired chains of events.

3.2.3 Syntax

The syntax of the language (here presented in Backus-Naur form [96]) is rather brief. It allows the declaration of entities as simple functional types (records, tuples, lists, or unions). Records may have fields. Rules contain expressions which have the typical shape of functional expressions in the study of ML languages, augmented with `wait`, `yield`, and queries[2] on lists:

Listing 3.2: Casanova 2 syntax

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

<moduleStatement> ::= module id
<openStatemnt>    ::= open id
<worldDecl>    ::= world id ["("<formals>")"] =
                   <worldOrEntityDecl>
<entityDecl>   ::= entity id ["("<formals>")"] =
                   <worldOrEntityDecl>
<worldOrEntityDecl> ::= "{" <entityBlock> "}"
<entityBlock>  ::= {<fieldDecl>} {<ruleDecl>}
                   <create>
<create> ::= Create "(" {<formals>} ") = <expr>
<formals>   ::= id [":" <type>] {"," <formals>}
<fieldDecl> ::= id [":" <type>]
<ruleDecl>  ::= rule id {"," id} "=" <expr>
<type>      ::= int |boolean  |float |Vector2
                |Vector3 |string |char
                |list "<" <type> ">" |<generic>
                |<type> "[" "]" |id
```

[2] To make high order functions (HOF's), such as map, or filter, simpler to write.

```
<generic>      ::= "'" id
<expr> ::= ...(* typical expressions : let, if,
                 for , while , new, etc. *)
          | wait (<arithExpr> | <boolExpr>)
          | yield | <arithExpr> | <boolExpr>
          | <literal> | <queryExpr> | <seq>
<seq>          ::= <expr> <expr>
<arithExpr>    ::= ...//arithmetic expressions
<boolExpr>     ::= ...//boolean expressions
<literal>      ::= ...//strings , numbers
<queryExpr>    ::= ...//query expressions
```

In the above we omit the trivial expressions grammar, which follows typical syntaxes such as in C#.

3.2.4 Semantics

The semantics of Casanova 2 are *rewrite-based* [62], meaning that the current game world is transformed into another one with different values for its fields and different expressions for its rules. Given a game world ω, the world is structured as a tree of entities. Each entity E has some fields $f_1 \dots f_n$ and some rules $r_1 \dots r_m$.

```
E = { Field_1 = f_1; ...; Field_n = f_n;
      Rule_1 = r_1; ...; Rule_m = r_m }
```

Each rule acts on a subset of the fields of the entity by defining their new value after a certain number of steps of the simulation. For simplicity, in the following we assume that each rule updates all fields simultaneously.

An entity is updated by evaluating, in order, all the rules for the fields, without guarantees about the order of execution:

```
tick(e:E, dt) =
 { Field_1=tick(f_1^m, dt); ...; Field_n=tick(f_n^m, dt);
   Rule_1=r'_1; ...; Rule_m=r'_m }
where
  f_1^m, ..., f_n^m, r'_m = step(f_1^{m-1}, ..., f_n^{m-1}, r_m)
  .
  .
  f_1^1, ..., f_n^1, r'_1 = step(f_1, ..., f_n, r_1)
```

We define the `step` function as a function that recursively evaluates the body of a rule. The function evaluates expressions in sequential order until it encounters either a `wait` or a `yield` statement. It also returns *the remainder of the rule body*[3], so that the rule will effectively be resumed where it left off at the next evaluation of `step`:

```
step(f_1, ..., f_n, {let x = y in r'}) =
  step(f_1, ..., f_n, r'[x:=y])
```

[3]Notice that this is not the actual implementation. Specifically, rules can only iterate through a finite set of subsets of its body, so we can just index (with an integer) the active subset, instead of representing the remaining code explicitly in memory.

```
step(f1, ..., fn, {if x then r' else r''; r'''})
  when (x = true) = step(f1, ..., fn, {r'; r'''})

step(f1, ..., fn, {if x then r' else r''; r'''})
  when (x = false) = step(f1, ..., fn, {r''; r'''})

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

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

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

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

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

3.3 Summary

The Casanova 2 language is an example of a domain specific language. Its goal is to abstract the logic of games by means of a series of primitives and language structures designed to capture properties shared among all applications in the domain of games.

Casanova 2 comes with syntax and semantics that are built around the domain of games. This results into games that are closer to their high-level descriptions. For example, see the short game described in Section 3.3.2, or the games listed in Appendix B.

In Section 7.2 an evaluation of our language is provided both in terms of performance and compactness of game code.

A language like Casanova 2 can only be called suitable for its purpose if it leads to an executable which has high performance. This means that it needs to be supported by a compiler that is able to create such fast executables. The compiler of the Casanova 2 language is discussed in the next chapter.

Chapter 4

Compiler architecture

In this chapter we discuss a concrete architecture of a compiler that implements the solution presented in Chapter 3. More precisely, we present a compiler that not only captures the syntax and semantics of the Casanova 2 language, but which also allows Casanova 2 games to run at high speed thanks to domain specific optimization. This chapter is divided into three parts: the first part discusses the compiler architecture and its internals (Section 4.1), the second part shows the specific generation of the most pervasive constructs, state machines, at a high performance (Section 4.2), and the third part shows to what extent the Casanova 2 compiler supports third party tools and engines (Section 4.3).

4.1 The structure of the Casanova 2 compiler

The syntax and semantics described in Chapter 3 are expressed in practice by means of a concrete architecture. This architecture comes in the shape of a source-to-source compiler [51]. More precisely the Casanova 2 compiler is a layered compiler, where every layer can be seen as a computational node, and is tasked with performing specific tasks. Besides capturing the syntax and the semantics of the language, the layers of the Casanova 2 compiler perform various kinds of transformations and checks, such as type checking, or generating the state machines for the bodies of the rules.

This structure is effective since it allows developers, who are developing a new feature of the language, to work only on one layer at a time, without breaking the others. Ideally, overlapping between layers should be minimal in order to keep the compiler, and its layers, maintainable. In Figure 4.1 a diagram shows the structure of the Casanova 2 compiler. In the figure, every box represents a layer performing a unique task. For example, the box `Code generation` transforms the abstract syntax tree (AST) into actual code, such as C# code. The "arrows" indicate the direction of the various transformations. For example, the arrow between the `Parser` and `Type checker` indicates that the parsed AST, which is output from the parsing phase, is given as input to the type checker; the type checker ensures that all the entities of the parsed AST respect the rules of the Casanova 2 type system. In the following, the various layers of Figure 4.1 are explained:

- Source code represents the source Casanova 2 game code.
- Parser transforms the game source code, which is given as a plain text file, into an abstract[1] tree instance, where each node of the tree denotes constructs occurring in the source code.

[1]The keyword abstract derives from the fact not every detail of the original syntax is represented explicitly. For example the expression (1 + 2) + 3 is represented by the node Plus(Plus(1, 2),3). Note that the parentheses are not mentioned. In this cases the parentheses are the omitted details.

- Type checker checks if the structure of the parsed AST is consistent with the rules of the Casanova 2 type system.
- Query optimization provides specific optimizations over typical predetermined query patterns. For example, a query that filters entities based on a predicate is transformed to perform the filter only if the entities populating the collection have changed state.
- State machine generation generates a state machine for every rule present in the source code.
- Semantics domain optimization provides some specific optimizations over typical structural patterns, which are common and legible into their more convoluted, but faster, equivalents.
- Code generation transforms the results of all steps into executable code. The language of the output is determined by the layer, therefore making the compiler adaptable. Moreover, this layer is responsible for adapting the resulting code, so as to make it work with a targeted framework, such as Unity3D. If no target framework is selected then an executable program (with a built-in game loop) is generated.

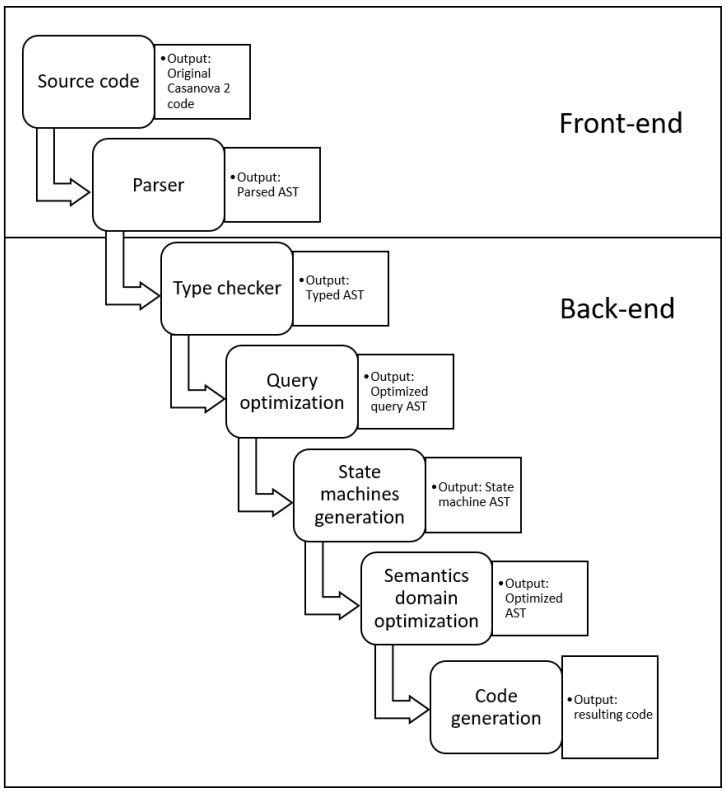


Figure 4.1: The structure of the Casanova 2 compiler

4.2 Code generation

Now that we discussed the shape of the Casanova 2 compiler we can discuss how Casanova 2 programs are executed by the machine. Every Casanova 2 program is converted into an equivalent one in C#.

Currently the code generation layer supports C#, but in the future we could easily support more languages, such as JavaScript, to run Casanova 2 directly in a browser.

The conversion of Casanova 2 constructs into a high-level, mature language, such as C#, is an important advantage: it prevents the problems associated with low-level languages, like Assembly, in the back-end phases of the implementation process.

Moreover, a significant advantage of using a language such as C# is that there are different compilers for C# that compile it to different platforms. This makes Casanova 2 programs portable, since we can use the .Net compiler or the `Mono` compiler to generate assemblies that can run on different platforms, for example .Net can be used to generate games that work on Windows machines, and same for Mono on Linux based machines.

In the following we discuss how Casanova 2 constructs are interpreted by our compiler in order to produce an equivalent version of them by means of a targeted language, in this case C#. In particular, we start this discussion with the data structures and functions that make up game entities, because entities represent the core of a Casanova 2 program, and they contain everything we find in a game: from the dynamics to the state (Section 4.2.1). The state, which is captured by attributes, is discussed in Section 4.2.2; the dynamics instead, which are captured by rule, are discussed in Section 4.2.3.

4.2.1 Entities

In Casanova 2, an entity can be a simple entity (in this case we denote the entity with the keyword `entity` next to the entity name), or the world entity (which is denoted with the keyword `worldEntity` next to the world entity name).

Both the world entity and the simple entities, are transformed into C# classes. For example the following Casanova 2 entities:

```
worldEntity Scene = {
  ...
}
entity Player = {
  ...
}
```

Are all translated into the following C# classes:

```
class Scene {
  ...
}
class Player {
  ...
}
```

Casanova 2 also supports inheritance, but only of Casanova 2 entities that inherit externally imported classes. This form of inheritance can be very useful when supporting third-party tools. It is common for game tools and engines to require users to inherit some external class, such as a class implementing a `MonoBehavior` of Unity3D.

4.2.2 Attributes

Every entity in Casanova 2 comes with a series of attributes that compose it. An attribute in Casanova 2 has a name and a type. The type can be primitive, such as integer, or string, or can be made custom (in this case a custom type can be either another Casanova 2 entity or an imported type, such a class

type from a third-party library). When transformed to C#, attributes in Casanova 2 are treated as C# attributes. In the following an example is provided, where an entity player contains three attributes: `Name` of type primitive `string`, `Position` of type imported `Vector3`, and `Faction` of type internal `Faction`.

```
entity Player = {
  Name : string
  Position : Vector3
  Faction : Faction
}

entity Faction = {
  ...
}
```

In this case the attributes of the entity `Player` are translated into attributes of a class `Player` in C#. The same holds for the entity `Faction`.

```
class Player = {
  String Name;
  Vector3 Position;
  Faction Faction;
}

class Faction = {
  ...
}
```

4.2.3 Rules

In Casanova 2 a rule expresses the dynamics of one or more entities, by affecting the state of their attributes. The rules of a Casanova 2 game are executed in order following a depth-first traversal strategy: starting from the world entity we first run in order (top-to-down) the world entity rules, and then we visit its attributes. For each attribute we first check if it is a Casanova 2 entity, if so then we visit its instance, run its rules in order (top-to-down), then for each of its attribute we repeat again the same behavior. We say that a *game iteration* is completed when starting from the world entity we stop the above traversal, because we reached the last Casanova 2 instance of the game tree structure. In Figure 4.2 we show a typical traversal of a Casanova 2 program, where four entities (`World`, `X`, `Y`, and `Z`) are traversed and updated. In this case a game iteration is complete after we run, in order, the rules (starting from the `World` entity): `W1`, `X1`, `Y1`, and `Z1`.

To avoid an instance to be traversed twice in the same frame, we use the keyword `ref`. The keyword `ref`, which is placed next to an attributes name, is used to denote a virtual reference to a Casanova 2 entity, the logical container of which is stored somewhere else in the game state. In the following an example is provided. In this example a player is stored logically in the attribute `Players` in the `worldEntity`, but in a turn based game we might need to track explicitly the current player. Thus, in `worldEntity` we also have an attribute `CurrentPlayer` that references the current player. Note the `ref` keyword next to `CurrentPlayer`; without it the current player instance would be updated twice per frame: once when traversing the players list and another one when traversing the current players attribute.

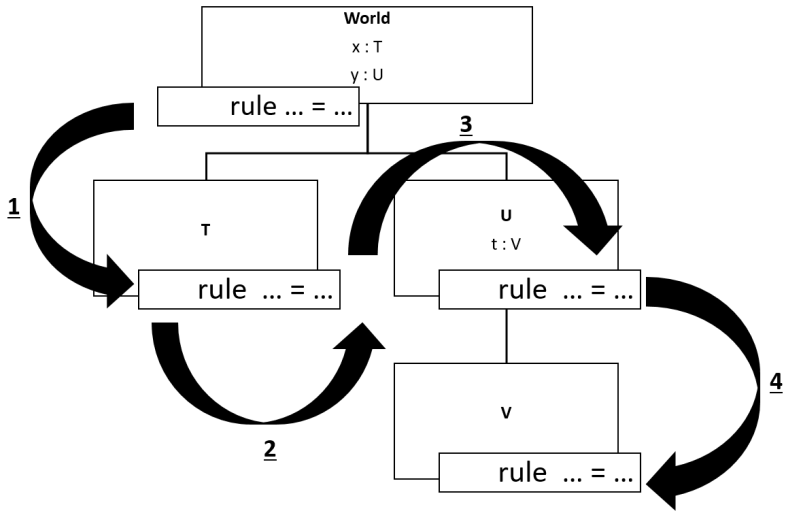


Figure 4.2: An example of a traversal in Casanova 2. In this case, starting from the `World` *we run the rules of each entity by following in order depth-first the arrows 1, 2, 3, and 4.*

```
worldEntity World = {
  Players : [Player]
  ref CurrentPlayer : Player
  ...
}
```

From the point of view of code generation, each rule is transformed into a method of the entity container class. Each transformed rule receives a unique id and is run in order by means of an update method belonging to the entity container class. In the following we show an example of a ball, to which a gravity force is applied. The entity comes with a series of attributes: `Position`, `Acceleration`, and `Velocity`. The force behavior is expressed by 4 rules: the first one (from the top) resets at every frame the `Acceleration` vector, the second one adds to `Acceleration` the force of gravity, the third one applies to `Velocity` the current acceleration, and the fourth one applies to `Position` the current `Velocity`. Note how the order of execution is very important. For example if we swapped the first two rules, then the ball would not move, since the acceleration would be equal to zero, when applied to the `Velocity`.

```
entity Ball {
  Acceleration : Vector3
  Velocity     : Vector3
  Position     : Vector3

  rule Acceleration = yield Vector3.zero

  rule Acceleration =
    yield Acceleration + Vector3(0f, -9.8f)

  rule Velocity =
    yield Velocity + Acceleration * dt
```

```
  rule Position =
    yield Position + Velocity * dt
}
```

When compiling the Casanova 2 program above, the compiler generates a class `Ball` containing the attributes given, but also adds multiple methods, distinguished by a unique id, one for each of the rules in the entity. Moreover, an additional method `Update` is also added to this class. When called, `Update` takes care of: (*i*) calling in order the rules of its instance, and (*ii*) traversing in order the attributes of its instance and calling, for each attribute of type Casanova 2 entity, its `Update` method. In case an attribute is a collection of Casanova 2 entities, then the `Update` method iterates each entity belonging to the collection, and for each these entities the corresponding `Update` method is called. If an attribute is denoted with the `ref` keyword, then the compiler will not generate any update call for it. In the following we show how the `Ball` program is compiled to C#.

```
class Ball {
  Vector3 Acceleration;
  Vector3 Velocity;
  Vector3 Position;

  public void Update(float dt){
    R0(dt);
    R1(dt);
    R2(dt);
    R3(dt);

    // Here we would, in order, iterate and update the
    // Casanova 2 attributes belonging to this class

  }

  public void R0(float dt){
    this.Acceleration = Vector3.zero
  }

  public void R1(float dt){
    this.Acceleration = this.Acceleration + Vector3(0f, -9.8f)
  }

  public void R1(float dt){
    this.Velocity = this.Velocity + this.Acceleration * dt
  }

  public void R1(float dt){
    this.Position = this.Position + this.Velocity * dt
  }
}
```

However, not all Casanova 2 constructs can be transformed directly into C# constructs, or with the minimal adjustments seen so far. In Casanova 2 each rule implements a series of instructions that

can be interrupted whenever necessary. This interruption happens only by means of specific language constructs, such as `wait`. Such constructs cannot be found in C# natively. However, by adopting special constructs for altering the execution flow of a Casanova 2 rule, it is possible at compiler time to break the rule into small sequential pieces, each representing specific actions to run at a specific time. This category of programs are typically referred as to *state machines* [45]. A big advantage of a state machine is that it allows to achieve high-performance, despite the fact that when building state machines, code typically loses important properties such as readability and maintainability. However, We can ignore this because the resulting code is not meant for developers to be consulted, but only for the machine to run. In the following we discuss how state machines are treated and transformed by the Casanova 2 compiler.

4.2.4 Generating state machines for rules' code

In accordance to the good and established practices of modern (object-oriented) software engineering, implementations of architectures similar to Casanova 2 for mainstream engines, such as Unity3D, are based on a series of nested state machines. Nesting allows some measure of separation of concerns and code reuse, and is therefore favored Unfortunately, nesting yields low performance because of repeated state selections, one per level of nesting.

In contrast, the Casanova 2 compiler produces an inlining of all the nested state machines into a single state machine with equivalent semantics, but faster runtime. The readability of the produced code is negatively affected due to the many low-level considerations, and the lack of structural nesting, which would otherwise help the reader with orienting himself in the original Casanova 2 code.

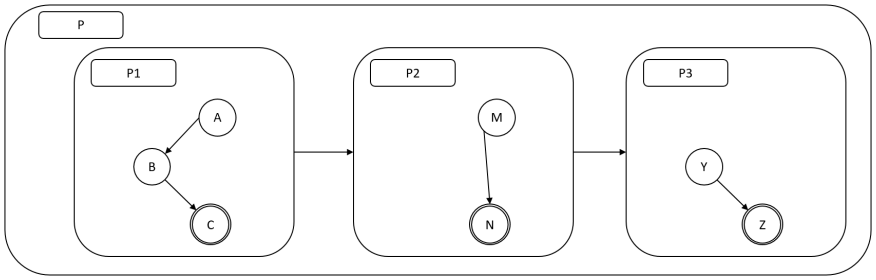
Flat vs non-flat state machines

In Figure 4.3 we show two equivalent state machines. The state machine in Figure 4.3a shows a non-flat state machine `P` which is made up of three inner state machines: `P1`, `P2`, `P3`. Each of them is running a series of operations, for example `P1` runs `A`, `B`, and then `C`. The order of execution is determined by the *arrows*. For example an arrow between `A` and `B` means: run `B` after `A` is done. When the source of an arrow is a state machine containing other state machines, then the source state machine must finish first with its internal logic, before continuing with the target of the arrow, for example, in our case the process `C`, which is marked with a double circle (meaning it is the last process of `P1`), is the last process to run inside `P1` before continuing with `P2`.

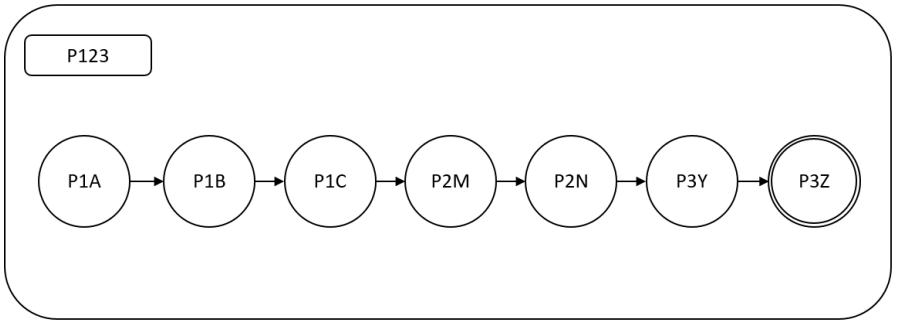
In the following a solution in pseudo code of the state machine in Figure 4.3a is provided. As we can see a series of variable (`StateP`, `StateP1`, `StateP2`, and `StateP3`) is used to track the state of the program `P` in order to select the current process to run. This allows the temporary suspension of the program, for example due to a `wait`, and its resuming, without messing up with its logical execution order.

Every process is run by means of the `run_program` function. The *run_program* function takes as input a program, such as `A`, or `B`, and a continuation. The continuation instructs the given program on how to behave when specific conditions are met. In our case all continuations are called after the current program is done. For example the instruction `run_program A (on done : StateP1 is 'B')` means: run program `A` and when `A` is done, set the state `StateP1` to `'B'`. In all our examples whenever a program ends, its continuation sets the next state machine to run by assigning the appropriate *state* variable. This simulates the "arrow behavior" described above.

This mechanism makes our formalism independent from any input program. Thus programs, such as `A` or `B`, can be made of either simple instructions, or complex programs using multiple state machines. What is common to all these programs is that once they are done, they all call a continuation that alters that flow of execution of the caller program.



(a) A non. flat state machine



(b) The flattened state machine

Figure 4.3: A comparison of a non-flat state machine and its equivalent one

```
switch StateP:
  case 'P1':
    switch StateP1:
        case 'A' -> run_program A (on done : StateP1 is 'B')
        case 'B' -> run_program B (on done : StateP1 is 'C')
        case 'C' -> run_program C (on done : StateP is 'P2')
  case 'P2':
    switch StateP2:
        case 'M' -> run_program M (on done : StateP2 is 'N')
        case 'N' -> run_program N (on done : StateP is 'P3')
  case 'P3':
    switch StateP3:
        case 'Y' -> run_program Y (on done : StateP3 is 'Z')
        case 'Z' -> run_program Z (on done : EXIT)
```

However, the above solution is expensive in terms of memory, since as the program scales in complexity, the memory needed to track the intermediate states increases as well. The same applies even more dramatically to CPU usage, since every `switch` requires the CPU to perform comparison operations which do not scale in terms of performance: more comparisons yield to more CPU load, and thus less performance.

Ideally we wish to have fewer variables and fewer state selections in order to improve the performance. More precisely, we wish to have one state machine, made of one switch, that captures the complete logical execution of the program. In Figure 4.3b an equivalent solution to the state machine proposed in Figure 4.3a is provided. Note that in this new solution we have only one state machine (`P123`) containing all the processes of the non-flat state machine, and where each process of the flat state machine is connected in the same logical order as in the non-flat version. In the following a code for this state machine is provided.

```
switch StateP123:
  case 'P1A' -> run program A (on done : StateP123 is 'P1B')
  case 'P1B' -> run program B (on done : StateP123 is 'P1C')
  case 'P1C' -> run program C (on done : StateP123 is 'P2M')
  case 'P2M' -> run program M (on done : StateP123 is 'P2N')
  case 'P2N' -> run program N (on done : StateP123 is 'P3Y')
  case 'P3Y' -> run program Y (on done : StateP123 is 'P3Z')
  case 'P3Z' -> run program Z (on done : EXIT)
```

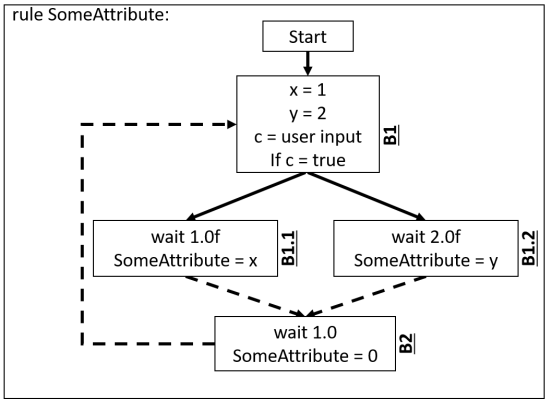
In the code above we managed to reduce the number of switches necessary to track the program state, without losing the logical execution order of the original program, since every process sets its continuation when it is done. This code is less readable, as the nested layers are not visible anymore, but it is faster to run. The compiler of the Casanova 2 language implements this second choice of state machine, which is implemented by means of code analysis. Our code analysis, inspects the game code to understand the nesting layers, and later uses this knowledge to implement a flat state machine that executes the intended original logic. In the following we provide a complete description of this optimization process.

Code analysis for state machines generation

At the compiler level every Casanova 2 rule block, which can be made up of different blocks with different levels of nesting, is interpreted as a control flow graph, which represents how the program control is passed between the various different blocks. Consider a Casanova 2 rule and its corresponding flow graph both depicted in Figure 4.4a and 4.4b respectively.

```
B1   rule SomeAttribute =
       let x = 1
       let y = 2
       let c = (bool) Console.Read()
       if c then
B1.1     wait 1.0f
         yield x
       else
B1.2     wait 2.0f
         yield y
B2     wait 1.0f
       yield 0
```

(a) A Casanova 2 rule; on the left its basic blocks are identified as B1, B1.1, B1.2, and B2.



(b) The flow graph of the Casanova 2 rule.

Figure 4.4: A Casanova 2 rule code and its control flow representation.

As we can see, every block in Figure 4.4a has an input and an output. For example, `B1.2` is reachable only by `B1`, and leaving `B1.2` can only bring the flow to `B2`. Note also the dashed arrows that appear in the flow graph. A dashed arrow between two blocks means that changing the control takes one frame to be completed. This behaviour corresponds in Casanova 2 to the `yield` statement, which suspends the execution of the rule, for one frame, right after updating the targeted attribute(s). By means of the flow graph analysis we can represent all possible nestings of rules and their relationships.

We track in a table information about the "follows" relationship between blocks. We can use this information to tell, when dealing with a sub-block, to whom it is supposed to return the control. For example, in Figure 4.4a `B2` is the block where any nesting after `B1` should eventually jump to. Thus, before compiling `B1.1` and `B1.2`, `B1` should inform the compiler that their exits must correspond to `B2`. We can apply this process recursively and to any level of nesting. As a result, every block is aware of which (other) block it is supposed to return the control to when it is done.

Moreover, the compiler always adds a last block called `EXIT_BLOCK`, which is always targeted when leaving the last instruction of the outermost block of a rule. `EXIT_BLOCK` points to the entry block of its rule. Block indexes are assigned incrementally, thus the first block is always indexed as 0 to force the rule to repeat itself after it is done. This allows interpretation of all instructions (without any modification) with the same algorithm, since the exit index is always provided as input to the recursive algorithm.

In the following, we provide the formal rules that show how the Casanova 2 compiler interprets blocks and generates the corresponding state machine. In general, an expression that has the following shape `[[expr]]_exit => CODE` means: transform `expr` to `CODE`; and use `_exit` as continuation to when `CODE` is done.

Moreover, in the following rules the keywords `goto` and `gotoSuspend` are used to simulate the temporal suspensions of Casanova 2 code. More precisely, a `goto X` instruction updates the state of the current block to execute of the current state machine to `X`, and jumps to the case of the state machine that corresponds to `X`; whereas `gotoSuspend` behaves similarly to `goto`, but instead of jumping we return the control to the caller of the method containing the state machine (in C# we simulate this behaviour with the `return` statement).

Interpreting a block of instructions We start with the rule that ignores the first simple expressions of a block followed by a non simple one. A simple expression is an expression that does not suspend the execution of the rule. Note that in the generated code we use `_lb*` and `goto`. `_lb` will become a new case of the flat state machine, while `*` denotes that `_lb` must be fresh and never used before (same applies for variables, in the following `int x*` stands for a fresh variable `x` with a unique name). The `goto` construct instead tells the program to jump to the label indicated by its value (in this case _lb_Es*).

The first simple rules of a block are collected into one case of the state machine, whereas the rest is reinterpreted by the compiler into a different case of the state machine. Note that when reinterpreting the `_exit` is kept the same, since all these expressions belong to the same nesting level.

`Rest` can be none, one, or more expressions (simple or not). We omit some trivial cases, such as when the block does not contain non-simple rules, or contains only simple rules, since they are trivial to implement and follow the same shape as specified below.

Listing 4.1: Interpreting block of instrutions

```
⟦SimpleExpression1
  ..
  SimpleExpressionN
  NonSimpleExpression
  Rest⟧exit
```

```
=>

_lb*:
  SimpleExpression1
  ..
  SimpleExpressionN
  goto _lb_Es*
_lb_Es*
  ⟦NonSimpleExpression
   Rest⟧exit
```

Interpreting a while loop A while loop generates a series of fresh labels, which are used to determine where to `goto` when evaluating the condition. If the condition is `false` then we exit the block, otherwise we continue with the body of the while. Note that when we interpret the body `B` we assign as exit to it the fresh `_lb*`. This means that when the body is done it will jump back to `_lb*` in order to perform the condition check again.

Listing 4.2: Interpreting a while loop

```
⟦while C do
  B⟧ _exit

=>

_lb*:
  if !C then
    goto _exit
  else goto _else*
_else*:
  ⟦B⟧lb*
```

Interpreting a collection Iterating a collection resembles a typical for loop iteration with an index variable, such as `for(int i = 0; i < ..; i++) { .. }`. Indeed we iterate the collection until the index has reached the end. Note, similar to the *while* loop, when interpreting the body `B` we assign to it as exit the label `_lb*`, where the block following `_lb*` performs the availability check of the next item. When all items have been iterated, control goes to `_exit`.

Listing 4.3: Interpreting a collection iteration

```
⟦for a in A do
  B⟧exit

=>

_for_lb*:
  var counter* = -1
  if A.length = 0 then
    goto _exit
```

```
  else
    var a = A[0]
    goto _lb*
_lb*:
  counter* ++
  if counter* >= A.length then
    goto _exit
  else
    a = A[counter*]
    goto _else
_else*:
  ⟦B⟧lb*
```

Interpreting an if-then-else Interpreting an *if-then-else* requires to generate two cases for the state machine: one that deals with the `then` body, and the other one with the `else` body. Both the *then* and the *else* blocks have `_exit` as label to relinquish control to when they are done.

Listing 4.4: Interpreting an if-then-else

```
⟦if C then A else B⟧exit

=>

_lb*:
  if C then goto _then*
  else goto _else*
_then*:
  ⟦B⟧exit
_else*:
  ⟦C⟧exit
```

Interpreting an if-then Interpreting an *if-then* requires to generate one state machine that deals with the *then* body. The *then* block has `_exit` as label to go to when it is done.

Listing 4.5: Interpreting an if-then

```
⟦if C then B⟧exit

=>

_lb*:
  if C then goto _then*
_then*:
    ⟦B⟧exit
```

Interpreting a wait with boolean condition A wait on a boolean condition keeps checking the condition until it becomes *true*. When *true* we jump to `_exit`. Note the `gotoSuspend` construct, which tells the state machine to resume from this block on next iteration, since the predicate is not *true*.

Listing 4.6: Interpreting a boolean guard

```
⟦wait CONDITION⟧exit

=>

_lb*:
  if !CONDITION then
    gotoSuspend _lb*
  else
    goto _exit
```

Interpreting a wait with a timer A wait on a timer counts down to 0 before moving the control to `exit`. As long as the timer is greater than 0 we keep decreasing it by `dt` and suspend the rule to let other rules perform their dynamics. Note that before starting to count down we first store the initial value of the timer in order to not let other rules interfere with this timer.

Listing 4.7: Interpreting a timer

```
⟦wait TIME⟧exit

=>

_lb*:
  var count_down* = T
  goto wait_lb*
_wait_lb*:
  if count_down* > 0.0 then
    count_down* -= dt
    gotoSuspend _wait_lb*
  else
    goto _exit
```

Updating an attribute When yielding (`yield`) we first update the attribute(s) which are affected by the rule in question, then we call `gotoSuspend` with `exit` as label to jump to at the next frame.

Listing 4.8: Interpreting a yield

```
[[yield E]] _exit

=>

_lb*:
  set E
  gotoSuspend _exit
```

Interpreting a rule The operation of interpreting the outermost block of a rule, denoted with the symbol [[...]], generates an extra label (our `_exit_block`) that is used to simulate the infinitely repeating loop behavior of a rule. Note that when interpreting the body of the loop we use the ⟦...⟧ operator that uses the interpretation seen so far.

Listing 4.9: Generating the exit_ block of a Casanova 2 rule

```
[[A]]

=>

_exit_block:
  [[A]]exit_block
```

However the above rule tend to generate a large number of labels and goto's which can be reduced in order to optimize the structure of the code. In the following we show two rules that are used in the compiler to reduce redundant labels and goto.

Optimizing goto The first optimization reduces the number of trivial goto's i.e. those goto's that jump to labels that are declared right after them. In this case the compiler traverses again the code in search of this pattern, and whenever we have a match the *goto* is removed. As a consequence when the control leaves that case it immediately falls into the next block.

Listing 4.10: Optimizing goto's

```
  A
  goto _lb
_lb*:
  B
=>
  A
_lb*:
  B
```

Compacting labels The second optimization performed by the compiler reduces the amount of consecutive labels into one. When compacting to one label, the compiler traverses also the sub-blocks in order to change all the goto using the old labels into goto that reference the compacted one.

Listing 4.11: Compacting consecutive labels

```
_lbx:
_lby:
  EXPR
=>

_lbxy:
  EXPR[_lbx ↦ _lbxy,
       _lby ↦ _lbxy]
```

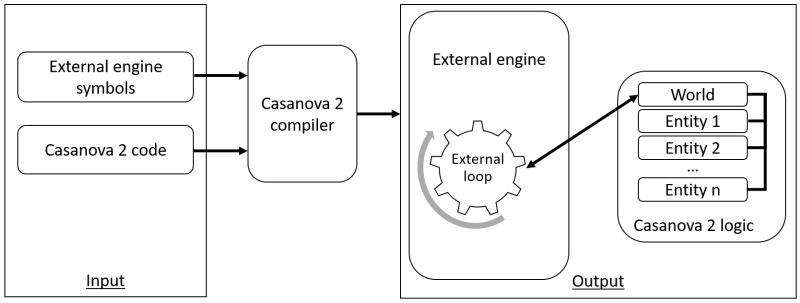
Discussion With the rules seen so far we managed to convert the body of rule into a flat state machine, disregarding its complexity and number of nesting. By manipulating only one state variable, and by using low-level instructions, such as `goto` that compile to very few machine operations (such as a single jump), we achieved a flat state machine that is computationally efficient. Naturally, the code maintainability of the generated sources is affected negatively, but this code is not intended to

be read by developers, and is sound because of the rules above: every expression above has only one unambiguous way of interpretation.

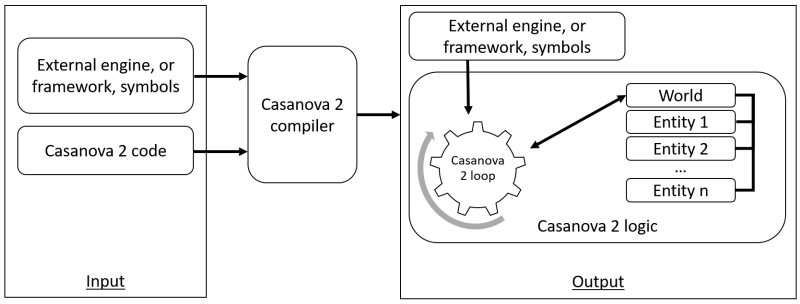
4.3 Supporting third-party tools and engines

Every entity has an update method, which is called by the entity that contains it. What about the world entity? Where is its update called? We can integrate the call of the world entity into a so called *game loop*. Such a game loop can be either provided externally or internally, as shown in Figure 4.5. In Figure 4.5 we show the only two types of game loops supported by Casanova 2: one provided internally and the other externally.

When the game loop is externally provided, then a layer is necessary that adapts a Casanova 2 game, to make it work with an external game engine (see Figure 4.5a). Otherwise, when there is not an external game loop, the Casanova 2 compiler generates an ad-hoc game loop that is independent from any game engine (see Figure 4.5b), which keeps polling the `Update` method of the world entity every 16 milliseconds. The choice of 16 milliseconds is due to the fact that there is no reason to run the game loop faster than 60 frames per second, as most monitors have a 60Hz of refresh rate. This parameter is easily configurable should there be reason for a higher framerate.



(a) In presence of an external game loop



(b) In absence of an external game loop

Figure 4.5: Representation of the input and output process of the Casanova 2 compiler w.r.t. the presence of external game engines or frameworks.

This mechanism also allows Casanova 2 to be, to some extent, independent of specific frameworks (it only depends on C#), since the language and its games are encapsulated and not aware of how they are used or where they are included. Indeed, the only adjustment a developer is supposed to

make, is to *teach* the compiler how to interpret the loop of each external platform, since every platform comes with a different way of dealing with the game loop: for example the loop of XNA/MonoGame is different from the one provided by Unity3D, and is yet again different from the one found in Unreal Engine. We implemented several examples of Casanova 2 applications that run on different platforms through the mechanism presented here, which we discuss in detail in Chapter 7.

Proxy system When interfacing with external libraries or frameworks, such as rendering or physics engines, in order to maintain the Casanova 2 source code independent from the details of these libraries, we need to use the so-called proxy system. The proxy system acts as an adapter between the Casanova 2 code and the external libraries and frameworks. The proxy system allows reuse of Casanova 2 code along with different libraries and frameworks. This way we can have the same game logic, written in Casanova 2, used in Unity3D, MonoGame, or Unreal Engine at the cost of minimal impact on the Casanova 2 sources. The proxies, each specific for one framework or library, will still need to be written by the developers in order to successfully connect the Casanova 2 code with the external library or framework.

In Figure 4.6 an example of a proxy is provided. In this example a Casanova 2 program imports an external `VisualPatrol`, which comes with two public members: `Position` and `Create`. Note that Casanova 2 is not aware of the concrete implementation behind the `VisualPatrol`, since in one case a concrete `VisualPatrol` is provided by a program that uses the facilities of XNA framework, whereas in the other case the program uses facilities from Unity3D.

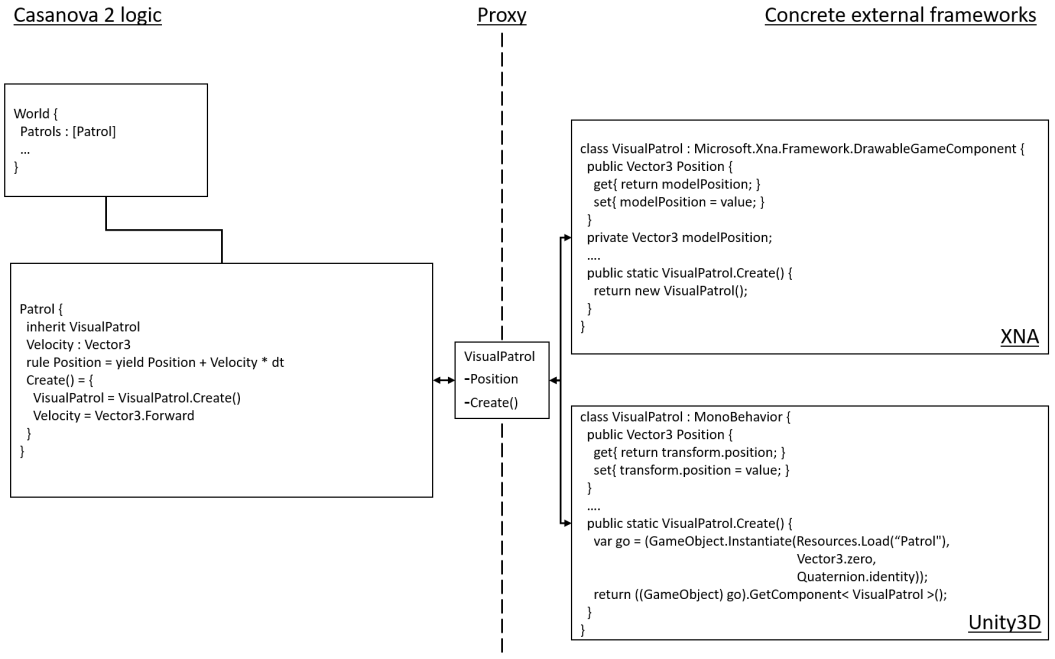


Figure 4.6: Casanova 2 code interfacing with two different frameworks, but both implementing the same proxy, namely `VisualPatrol`.

A proxy system is meant to generalize over the aspects of the target library or framework that are needed for the Casanova 2 program. This then ensures that the Casanova 2 program can remain the same, regardless of the external system used. Examples of concrete proxies interfacing

a common library for game development can be found at https://github.com/vs-team/casanova-mk2/wiki/Casanova%20Proxy.

4.4 Summary

In this chapter we discussed the implementation of a compiler for the Casanova 2 language. In particular, we discussed the structure of this compiler, and showed how this compiler interprets Casanova 2 programs, so as to generate code that exhibits fast runtime performance. Moreover, we discussed how Casanova 2 programs interoperate with third-party tools and engines, by means of the so-called proxy system. In next chapter, we will investigate further the opportunities offered by the domain of games to improve even even more the performance of Casanova 2 programs.

Chapter 5

Compiler optimization

In this chapter we show a case study on how to improve the performance of the Casanova 2 language runtime by means of domain specific optimization. Domain specific optimization arises from a series of observations about recurrent patterns in code that, while being idiomatic and frequently used by programmers in practice, exhibit undesirable runtime properties (i.e. they are too slow or use too much used memory). The domain specific optimization is applied in the form of a series of heuristics to recognize such idiomatic code and transform its semantics, to alleviate the negative properties. The domain specific optimization thus allows programmers to write clear, readable, intuitive, idiomatic code, but with the same desirable performance as hand optimized code (which is far more complicated to handle). A typical example of domain specific optimization is found in the SQL family of languages.

Specifically, in this chapter we present a solution to the loss of performance in games that occurs as a consequence of the encapsulation design pattern, which is generally used to keep code maintainable.

5.1 Maintainability vs. speed

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 the game complexity increases, this leads to an increase in size and complexity of the components, which, in turn, leads to an increase in the complexity of developing and maintaining them, and thus an increase in development costs.

Since a video game, during its development, is in a continuous evolution, it is often the case that at the end of its development the final design of the game is quite different from the initial design. If not tackled in advance this evolution will affect heavily the available resources, as non maintainable game code will require considerable development time to be fixed and adapted to design changes. To alleviate such costly changes, game code should be structured in a way that maintaining or restructuring it is relatively painless.

According to [16], the typical life cycle of software implemented by means of a programming language is: (***i***) *building a prototype*; (***ii***) *designing* a version of which code is readable and maintainable; and eventually (***iii***) *optimizing* (after obtaining confidence with the context and the problem) the code from the previous point, to meet any remaining (often non-functional) requirements.

We can see that this cycle is applicable to game development as well: (***i***) building a *game prototype* is always necessary to become confident 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; and (***iii***) *optimizing* is

a common process in game development, for example in the case of performance optimization (which is of high importance for games).

Regarding designing maintainable game code, this is usually done using software development techniques. Software development techniques have been studied to improve software maintainability and tackle complexity [30]. Encapsulation, which consists of isolating a set of data and operations on those data within a module and providing precise specifications for the module [56], is an example of a technique aimed at increasing code maintainability and readability

Indeed developing a game is a highly dynamic process [99] involving a wide variety of team members with different roles, such as designers, programmers, artists, 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 [72]. Small changes to the design translate into considerable amount of code. For example, since a game may feature many small entities, encapsulation forces those entities to interact through specific interfaces. In Figure 5.1 we see an example. In the upper part of Figure 5.1 the class `A` has an explicit reference to an instance of type `B`. This means that `A` can interact with, and know everything about, the internals of `B`. This entails that whenever a part of B changes, if A uses that part as well, A needs to change too. In the lower part of Figure 5.1 an equivalent version to the first one is provided, but in this case the shared aspects have been encapsulated, and are provided by means of an interface, to `A`. This means that A only knows how to access the aspects of B it needs via the interface IB, and whenever `B` changes, as long as the interface `IB` does not change, `A` does not need to be notified. Moreover, this mechanism allows the definition of different implementation of `IB` (see `B1` and B2), which can be used in different situations, without the necessity to change the code of `A`.

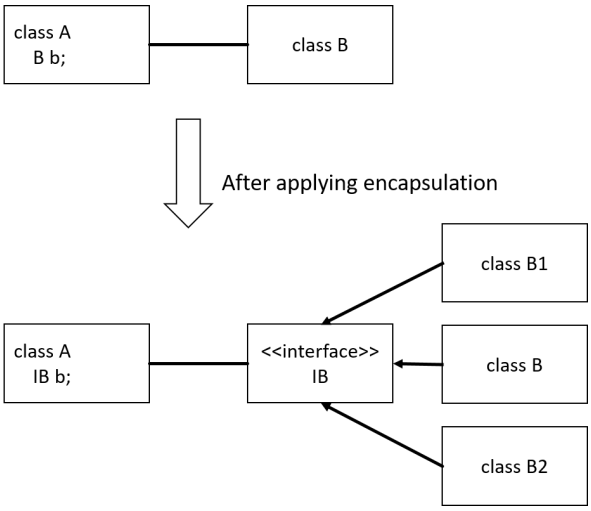


Figure 5.1: The two diagrams show two versions of code structures: one which does not use encapsulation, and the other one which uses encapsulation.

When calling methods of the interfaces, overhead is added due to dynamic dispatching [105]. 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 [7].

What seems ideal is to have the advantages coming from both stages (*ii*) and (*iii*): game code that is well maintainable and readable, while at the same time being fast at runtime. To this purpose,

we investigated this problem and developed a solution that allows developers to write encapsulated code in Casanova 2, which through extensive automated optimization turns source code into a high-performance executable, thereby relieving developers from refactoring design structures by hand, thus reducing the chances to make mistakes and the overall game development costs.

We start with a discussion on the focus of this chapter and related works (Section 5.2). Then we discuss encapsulation and typical complex optimizations, which break encapsulation, by introducing a case study. We use the case study to identify issues in using both the encapsulated and the optimized code in Section 5.3. We then introduce our idea for dealing with encapsulation without losing performance Section 5.4. We use this idea to propose a concrete implementation, with corresponding semantics, within the Casanova 2 language in Section 5.5. Eventually, in Section 5.6 we discuss the advantages of our technique.

5.2 Focus of the work and related works

The focus of this chapter 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 optimization problem, but also severely constrains the spectrum of possible solutions. To understand this, consider that on a 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 frames are regularly 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.

The two frameworks that, however, are potentially suitable for our purpose of optimizing the speed of game code (while still retaining encapsulation) are runtime dynamic machinery, and compile-time code generators.

5.2.1 Runtime dynamic machinery

Highly dynamic frameworks typically make use of mechanisms that either feature large numbers of dynamic/virtual calls, or rely on reflection. The use of dynamic/virtual calls within a big hierarchy of objects has a dramatic impact on performance because it severely disrupts cache coherency [101]. 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 [83]) 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 aspects 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.

5.2.2 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. 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 [58]). 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 at the (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 [101].

Games choose runtime dynamic machinery via mostly object oriented design patterns, and reflection when strictly needed. In the following section we discuss a short example to explain the problem of encapsulation in games, and in the end we discuss the advantages and disadvantages of using encapsulation when designing a game.

5.3 Encapsulation in games - an 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]).

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.

5.3.1 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 traveling along its incoming routes, which are close to the planet, and moves them into the attacking fleets list;

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)
```

5.3.2 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 [91], 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 [36].

As far as *performance* is concerned, in the encapsulated version, the planet queries the game state to obtain all routes of 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 that 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. One could argue that adding a spatial index 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.

5.4 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. In particular, we will discuss the idea behind how to optimize the lookups of those elements in the game that exhibit some specific temporal behavior. Moreover, we will discuss where to implement such optimization.

5.4.1 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 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 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.

5.4.2 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.) [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 blocks of code, and apply an optimization that (*i*) keeps their code block inactive in a *sleeping* 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.

5.4.3 Language level integration

The process described above can be automated at the compiler level as a 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 that are suitable for this optimization. The Casanova 2 compiler optimizes the encapsulated implementation by creating and maintaining the required indices. This way the code written by the programmer will keep the benefits of readability and maintainability that encapsulated code holds, 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.

5.5 Implementation Details

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 with the supporting data structures, and to achieve better performance, we optimize those predicates that exhibit temporal locality, selected based on manual annotations.

We will refer to a predicate on fields that exhibit temporal locality as *Interesting Conditions* (ICs). These predicates are stored in a data structure called the *Interesting Condition Data Structure* (ICDS).

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 5.2.

5.5.1 Casanova 2 rule

In Casanova 2 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.

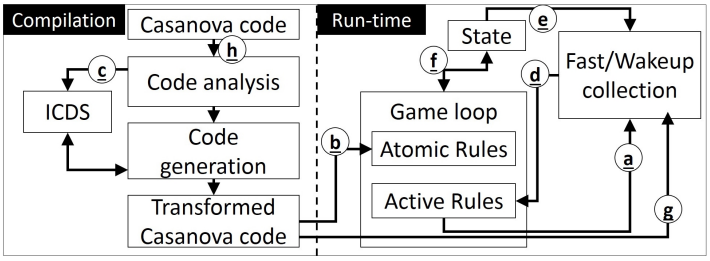


Figure 5.2: System Configuration

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
  wait ShieldReloadTime
  yield 100
```

5.5.2 Compilation - Recognizing ICs in Casanova 2

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 5.2(h)), which requires the identification of ICs in code. This is discussed next.

Casanova 2 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 2 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 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 traversing 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 ← ∅
    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 ∪ {(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 ∪ {(e, r.index, r.domain, p)}
                end if
            end if
        end for
    end for
    return ICDS
end function
```

Given a Casanova 2 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 5.2(c)).

5.5.3 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 5.2(b)) that are run every frame, and suspendable rules (see Figure 5.2(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 2 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 5.2(f)), we activate the corresponding *Interesting Suspendable Rule* (ISR) `R` by triggering it (see Figure 5.2(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 to one of these dictionaries as a *Dictionary of Entity-Predicates* (`DEP`).

We use the static information from the ICDS (see Figure 5.2(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)`.

5.5.4 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 `OBS` pairs, 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;
- **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 5.2(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 2 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 5.2(a)).

5.5.5 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.

5.5.6 Examples

In the following we present three code snippets, and discuss briefly how they are interpreted by the aforementioned approach.

Example 1 The first snippet below, a suspended rule update, presents the entity `E`, which contains a rule that waits until the condition `C` become `true` and a rule that updates `C` every five seconds. `C` is an interesting condition and changes only occasionally, thus the associated rule, which updates F, can benefit from optimization.

```
entity E =
    F  : T
    C  : bool
    rule F =
        wait C
        B
    rule C =
        wait 5.0f
```

```
        yield not C
```

Example 2 The second snippet, an atomic rule update, behaves similarly to the previous one, except that `C` changes every frame. In this case `C` is an interesting condition, but the rule that changes `F` will not benefit from the optimization as `C` changes constantly.

```
entity E =
    F  : T
    C  : bool
    rule F =
        wait C
        B
    rule C = yield not C
```

Example 3 In the third snippet, a suspended query rule update, `F` is updated by selecting those elements in `Elems` (a collection of elements of type `S`) that satisfy a condition `C`. `C` is a field in `S` which changes every five seconds.

```
entity E =
    F     : [T]
    Elems : [S]
    rule F =
        [for e in Elems do
         where e.C
         select e]

entity S =
    C : bool
    rule C =
        wait 5.0f
        yield not C
```

Our compiler analyses the query above so as to generate a rule in `S` that adds `this` to the collection `Elem` in `F`, but only when the condition `C` is `true`. If the value contained in `C` exhibits some temporal locality then the compiler will optimize the new generated rule so as to check the value of `C` only when `C` is updated.

```
entity E =
    F     : [T]
    Elems : [S]

entity S =
    ref SourceE : E
    C     : bool
    rule SourceE.F =
        wait C
        yield this @ SourceE.F
```

5.6 Summary

Game developers often have to choose between maintainability of their code and speed of execution, a choice that more often than not flavours 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. In this chapter, we presented an implementation of this solution in the Casanova 2 language. In Section 7.2 an evaluation of Casanova 2 is provided both in terms of performance and compactness of game code.

Chapter 6

Designing games with Casanova 2

In this chapter we discuss how to design and make games in Casanova 2. We begin with an introduction of the basis ingredients for making games in Casanova 2 (Section 6.1). These ingredients are meant to introduce the fundamental elements that are necessary when designing a game in Casanova 2. We then use these elements to describe a general design for building real-time strategy games (Section 6.2). Eventually, we use this design to implement a concrete real-time strategy game (Section 6.3).

6.1 Casanova 2 games basis ingredients

Casanova 2 is a language designed to capture common aspects of video games by providing domain specific constructs. By composing these constructs it is possible to encode different programs for different video games.

The space of encodable programs is gigantic in scope, and a programming language typically narrows this scope by coming up with a limited amount of constructs and their combinations. This narrowing function of programming languages is needed to make the problem of encoding programs tractable in practice by human programmers, but of course it comes at a cost. While some programs become easy and clear to express in a given language, many more programs cannot be encoded easily and sometimes cannot be encoded at all. A language such as Casanova 2 covers well the expression of some specific programs, but does so at the expense of others.

We now turn our attention to informally estimating what sort of narrowing function is performed by Casanova 2 on the domain of encodable programs. This means identifying the idioms that Casanova 2 proposes and implicitly tries to enforce. These idioms encompass all those programs and functionalities that are "easy to write". The collection of these idioms defines a path of least resistance for making games in Casanova 2, leading to fundamental design guidelines of games built with the language. The idiom of Casanova 2 is based on the fundamental concepts of entities, attributes, and rules. Entities and attributes represent the static structure of a game, i.e., the rigid elements of a game that do not change during its life time. Rules represent the dynamic parts of a game, i.e., the moving parts of a game that continuously change during its life time.

6.1.1 Entities

In Casanova 2, the scene of a game, which is made up of a structured collection of elements, is captured by the so-called *game entities*. Game entities are always grouped into a graph structure, the entry point of which is an entity called `worldEntity`. We can think of the `worldEntity` as the entry point of a game, which contains all the **visible** and **abstract** elements of the game. **Visible** entities

are the concrete parts of the game, which typically appear in the early stages of the game design. Among these visible entities we can find entities such as a soldier, a rock, a car. **Abstract** entities are those elements in a game that we typically cannot see, since they work in the background. For example, an abstract entity might be used to perform some background operation, such as a battle entity instantiated to referee a fight between two players. Abstract entities are less intuitive than the visible ones, since they might not be explicitly part of the design itself.

In the following code listing we introduce a game, which is made of an outer container `Galaxy` (the root of this game structure), a visible `Player` entity, a visible `Planet` entity, and an abstract `Link` entity, which will be used to connect two different planets (in this case a player can travel the galaxy only through links).

```
worldEntity Galaxy = {
  ...
}

entity Player = {
  ...
}

entity Planet = {
...
}

entity Link = {
...
}
```

6.1.2 Attributes

Every entity is made of a series of characterizing attributes that specify how data is stored. When an entity is instantiated, space is reserved in memory to store the values of its attributes. The values of the attributes of an entity are called *state of the entity*. The *state of the game* is then the union of all states of all entities populating the game.

Every attribute can either be a primitive value, such as a number, or a reference to a derived type, such as a Casanova 2 entity or an external imported class. Primitive types are typically atomic, whereas derived types are compositions of other types.

When referencing a Casanova 2 entity, the developer must distinguish whether the attribute is a foreign or a primary reference. Primary references represent the "is composed of" relationship, whereas foreign references represent the "knows about" relationship. This distinction is important for the compiler, to avoid an entity to be updated more than once per frame in the presence of cyclical references, and for the developer, to distinguish dependencies and structure. By default all attributes in Casanova 2 are primary. If a developer wants to declare an attribute as foreign then he has to add the keyword `ref` next to the attributes' name. As in the database literature [34], a foreign reference is used to declare a *weak* link between two entities, whereas a primary reference is used to distinguish unambiguously an entity from another. For example the position of a planet is primary, whereas its owner could be foreign, since a planet can be owner-less.

Attributes are the only way in Casanova 2 to simulate containment of entities (if B is logically contained in A, then A must have an attribute of type B). Moreover, by default every entity (except for the world entity) has a foreign and implicit attribute that references the *world* entity, since referencing

the *world* entity explicitly is not possible in Casanova 2. This constraint enforces the meaning of the *world* entity, which represents something intrinsic and always available in the game to every other entity.

In the following code listing we continue building upon the example given in the previous section to show how attributes can be used to determine the structure of entities and their logical organization. In this example the player(`Player`), the collection of available planets (`[Planet]`), and the collection of the allowed links between planets are all stored in the `Galaxy` world entity.

Every planet has at least an optional `Owner`, a `Name`, and a `Position`. When conquered, a planet changes its owner, to reference the player who just conquered it. Note that `Owner` in `Planet` is marked as `ref`, which means that `Owner` is a foreign attribute, thus is just used to establish the link between the player and the planet, because the owner is not a part of the planet itself.

Every link is directed and connects two planets by two foreign references, each per planet.

```
worldEntity Galaxy = {
  MainPlayer : Player
  Planets    : [Planet]
  Link       : [Link]
  ...
}

entity Player = {
  Name : string
  ...
}

entity Planet = {
  ref Owner : Option<Player>
  Position  : Vector3
  Name : string
  ...
}

worldEntity Link = {
  ref From : Planet
  ref To   : Planet
  ...
}
```

As for entities, attributes not only capture **visible** aspects of a game, but also the **invisible** or **abstract** ones. An example of a visible aspect of a game, which is captured by an attribute, is the number of current lives of a player. The number of current lives of a player is typically displayed on the screen and stored inside the players entity.

```
...
entity Player = {
  Lives : int
  ...
}
```

Another example of a game aspect captured by an attribute, but this time abstract, is the time it takes for a player's ship to move from one planet (`CurrentPlanet`) to another (`Destination`), after

the engine is started (when `EngineStarted` is `true`). In this case, the attribute (`TimeLeftToArrive`) is an abstract concept, internal to the ship, meant to internally simulate the travelling time.

```
...
entity Ship = {
  EngineStarted : bool
  TimeLeftToArrive : float

  ...

  ref Destination    : Option<Planet>
  ref CurrentPlanet  : Planet

  ...
}
```

6.1.3 Rules

So far we discussed the capturing of game elements, by means of entities, and how to fill them with data, by means of attributes. Without any sort of dynamic logic the game state will never change, thus it would keep the same values for ever. But a game is a dynamic system where all entities move, and interact with each other. Thus, to achieve such dynamism we need to change the state of the entities populating the game.

In Casanova 2, the state of an entity can only be changed by means of a rule. Indeed, rules are the only machinery able to perform the dynamics of a game. Every rule belongs to one instance of a game entity, and is defined inside the entity declaration of such instance. Thus, the moment an instance disappears from the game state, the associated rules will stop affecting the game state. A rule has a limited effect, which means that it can change limited portions of the game state. Indeed, when defining a rule we must also declare what attributes of what entities are affected by the rule.

The dynamics of a rule are expressible via a block of code, which can be executed either **atomically**, or **discretely**.

When **atomic**, the code of a rule is executed all at once without interruptions. Continuous rules capture those aspects of a game that are always true and cannot be interrupted, such as gravity, or the current financial value of a city in a city simulation game.

In the previous example, we could define a rule that applies continuously a velocity to the ship's position. In this case the rule affects the `Position` attribute of its instance (our ship) by applying, every frame, the current `Velocity`.

```
...
entity Ship = {
  ...
  Position : Vector3
  Velocity : Vector3

  rule Position = yield Position + Velocity * dt

  ...
}
```

When **discrete**, the flow of a rule is not continuous. Thus, it can be interrupted, for example to wait for an external condition to happen. Discrete rules capture those aspects of a game that takes a long time and various real-time decisions to complete, and thus need intermediate steps, the completion time of which may be unknown or might take some time, to be completed such as spawning an entity, a timer, or behaviors that require synchronization between two different entities.

Be means of discrete rules we are able to define a rule that waits for the ship to have a targeted planet, and once the engine is started it sets up the velocity and the arrival timer. Once the ship is arrived the timer and the velocity are set to zero. Note the second discrete rule, in the following code listing, which is working in strict synchronization with the first one. This rule keeps decreasing the arrival timer by `dt` whenever the arrival timer is greater than zero (this rule is actually simulating a countdown behavior). Without this rule, the first discrete rule would end up in a starvation situation.

```
...
entity Ship = {
  EngineStarted : bool
  TimeLeftToArrive : float
  ref Destination    : Option<Planet>
  ref CurrentPlanet  : Planet
  Velocity : Vector3


  rule TimeLeftToArrive, Velocity =
    wait EngineStarted && Destination.IsSome
    yield GetTime(Destination.Value, CurrentPlanet),
          GetVelocity(Destination.Value, CurrentPlanet)
    wait TimeLeftToArrive < 0
    yield 0, Vector3.zero

  rule TimeLeftToArrive =
    wait TimeLeftToArrive > 0
    yield TimeLeftToArrive - dt

  ...
}
```

So far we have seen the fundamental idioms that compose any Casanova 2 game. By composing them and following the good practices suggested above, we can build different kinds of games. In order to show the quality of our idioms and their generality in practice, we now show a concrete example of a game design captured by means of our Casanova 2 idioms. More precisely, we will discuss the real-time-strategy (RTS) game genre, and will provide a concrete example of its implementation.

6.2 Building RTS games in Casanova 2

In the video game industry real time strategy (RTS) games are one of the most popular genre [28]. Moreover, RTS games are used as frameworks for many different kinds of serious scenarios, such AI simulations [23], simplified military simulations [22], and learning [86].

Thus, because of its relevance, especially for the serious games scenario, in this section we will focus our attention on the development of games belonging to this genre in Casanova 2.

We will use this section to show to what extent the idioms presented in the previous section are

good at capturing complex game designs. We will first discuss a general taxonomy for RTS games (Section 6.2.1) and show how Casanova 2 idioms cover such taxonomy (Section 6.2.2). Then we will use the implemented taxonomy to build a concrete game (Section 6.3).

The game discussed in this section is based on the design of an already existing video game called *Galaxy Wars*, a real-time space strategy game. This design of such game is discussed in Section 6.3.

6.2.1 An analysis of RTS games

Implementing an RTS requires writing code for all of the common game elements such as units, battles, movement, production, resources gathering, statistics, etc.

We identify the common game elements of an RTS by mean of a taxonomy [4]. In this paper a design pattern which we call RAA[1] (resource, actors, action) is introduced for representing RTS'. In particular the design effectively describes any RTS game in terms of:

- *Resource*, which is any kind of game statistic. A statistic might represent a numerical value of a battle, or the cost to deploy a unit, etc.
- *Actor*, which is any kind of game element that contains resources. We distinguish different entities by their resources and actions.
- *Action*, which describes an interaction, is used to describe the flow of resources among entities.

Whereas the definition of action given above covers generic types of interactions (like the attack of a ship, or the percentage of construction) special attention should be given to the specific sorts of actions that are common to all RTS games. We identified these special actions in terms of: *creation*, *deletion*, and *strategy update*:

- *Creation* An entity is created after some conditions in the game world are met. A condition could be for example the player who decides to create a fleet to attack an enemy player, an automated spawner that after a certain amount of time creates a unit, etc. Furthermore, the creation of an entity typically consumes some game resources of the player. If the resources are not enough then creation will be postponed or not allowed at all.
- *Deletion* Analogous to creation, an entity is deleted after some conditions in the game are met. A condition could be for example during a battle the life of the entity is lower or equal to zero. Entities removed from the game world are not able to interact with other entities.
- *Strategy update* During the life time of an entity it often happens in an RTS that the entity changes its behavior. For example a resource gatherer unit mainly collects resources, but if necessary it can also attack; a fleet moving around the world might eventually end up in the local fleets of a planet or take part in a battle. All these actions differ from each other, indeed their logics affect different sets of resources even though the entity remains the same.

Next, we discuss how express the just introduced taxonomy in the Casanova 2 language.

6.2.2 Abstracting RTS games in Casanova 2

In this section we show how Casanova 2 can implement the RAA pattern, and also extend it with the special actions (CDU): creation, deletion, and strategy update. More specifically, for each element of this pattern we will provide an abstraction that captures it by means of some Casanova 2 constructs.

[1] In the original manuscript this taxonomy is called REA [4], but to avoid ambiguity with the definitions of Casanova 2, we will refer to resources as resources, entities as actors, and actions as actions.

Resources

Resources can be modelled as a Casanova 2 entity with no rules, and a field for each of the resources used in the RAA pattern. In a game we might have different resource entities for different groups of resources. We define a resource entity by first defining its name `ResourceName` and then by listing the resources contained in it. A resource in Casanova is a field and it is defined as a tuple `Resource * Type` where the first item refers to the field name while the latter refers to the field type. In the following code listing we show a generalized description for a generic resource entity.

```
entity ResourcesName =
  R1 : T1
  R2 : T2
  ...
  Rn : Tn
```

Actors

RAA entities can be modelled directly as Casanova entities. An actor will contain the `Resources` (of type ResourcesName) and a series of rules that will act as the constant, mutable, and threshold actions of RAA.

```
entity Actor =
  Resources              : ResourcesName
  //constant actions
  //mutable actions
  //threshold actions
```

Actions

Following the RAA pattern, we divide the actions into 3 categories: (*i*) constant transfer, (*ii*) mutable transfer, and (*iii*) threshold transfer. We model RAA transfers as rules in Casanova.

An action in RAA simply connects a source and a target. In Casanova the source is the action/rule container while the target is an entity containing a field that refers to the source. The target checks the source reference whenever it needs to interact with it. More specifically, the source resources are read by the target actor periodically to locally update their fields. The resources to transfer generated by actions are stored inside the source entity of the same actor. We refer to the resources to transfer as `Transfers` in Casanova. The definition of the actions will be shown in the following items.

- **Constant transfer** A constant transfer simply *adds* the resources of the source actor to the resources of the target. The following rule, which is contained in the source actor, updates the `Transfers` whenever a condition is met, `restrictions` is a predicate that specifies a condition to apply the action. The rule waits one frame, to ensure that the target actor reads the change, before resetting the `Transfers`.

```
enity SourceActor =
  Resources   : ResourcesName
  ...
  rule Resources.Transfers =
    wait restrictions
    yield Some(some_resources)
    yield None
```

Every time some `Transfers` are produced the target actor reads them and updates its resources accordingly. We use the same `restriction` as in the source entity to ensure that the generated `Transfers` belong to that specific target instance. We assume for brevity that we have a + operator for the entity `Resources`, which behaves like a vector sum, to be used by the aggregate function `sum` in the query.

```
enity TargetActor =
  Resources   : ResourcesName
  ref Source : SourceActor
  ...
  rule Resources =
    wait Source.Transfers.IsSome & restrictions
    yield Resources + Source.Transfers.Value
```

- **Mutable transfer** In the mutable transfer the resources are *moved* from the source to the target. A transfer can be also negative in RAA. In case of negative transfers we simply swap the logic so the source implement the behavior of the target and vice-versa. The rule of the mutable transfer behaves almost the same as for the continuous transfer. The only difference is that in the source together with setting the `Transfers` by an amount `some_resources` we also remove the same `some_resources` from the source resources. Again, we assume for brevity that we have a - operator for the entity `Resources`, which behaves like a vector difference, to be used by the aggregate function `diff` in the query. In the following we use a `Resources\{Transfers}`; this is shortcut to say select all the attributes in `Resources` which are not in `Transfers`.

```
enity SourceActor =
Resources   : ResourcesName
...
rule Resources.Transfers, Resources\{Transfers} =
wait restrictions
yield Some(some_resources), Resources\{Transfers} - some_resource
yield None
```

- **Threshold transfer** The threshold transfer is a constant or a mutable transfer that executes the resources transfer, as in the examples above, *until* a certain `threshold_condition` is satisfied. Once we meet the `threshold_condition` a series of output values are yielded and then reset. For this kind of action we need to extend the source entity definition with additional fields to store the output of the rule.

```
enity SourceActor =
Resources   : ResourcesName

Output_0 : Option<T_0>
Output_1 : Option<T_1>
...
Output_n : Option<T_n>

...
rule Resources.Transfers, Output_0, ..., Output_n =
.| threshold_condition ->
yield None, Some value_0, ..., Some value_n
```

```
yield None, None, ..., None

.| _ ->
wait restrictions
yield Some(some_resources), Output_0, ..., Output_n
yield None, Output_0, ..., Output_n
```

Creation

Creation of an entity always follows some event. In Casanova we can combine the creation expression with action. This is allowed since inside a rule in Casanova statements are run imperatively. The following shows a generalization for the creation of an object after an action is run. An entity of type `SomeEntity` is spawned after an action is run.

```
enity SourceActor =
  OutputObject : [SomeEntity]
  ...
  rule Resources.Transfers, SomeObject =
    // an action
    yield Resources.Transfers, [new SomeEntity(some_parameters)]
```

Deletion

If an instance `O` is about to get destroyed, all instances `I`'s that share some logic with `O` must be notified that `O` is about to get destroyed. An instance of `I` knows that `O` is about to get destroyed when `O` is moved into a special field called `DestroyedO`. `O` is moved into `DestroyedO` for a certain amount of time before we reset the `DestroyedO` field. In the following code `SourceActor` contains, besides the usual fields, also a reference to an object `O` of type `Object` and the `DestroyedO`, which is an option of type `Object`. Below, `Option<T>` is either one `T`, or none. It is a safer alternative to nullable values, coming from the world of functional programming.

```
enity SourceActor =
  ref DestroyedO : Option<Object>
  O : Option<Object>
  ...
  rule O, DestroyedO =
    wait restrictions
    let acc = O
    yield None, Some acc
```

Strategy update

An entity moves according to some logic. In this case we can apply a constant transfer to for example update an entity position according to its velocity. An entity might change its behavior according to some conditions. For example a fleet might change from travelling to attacking. This kind of behavior might resemble the strategy pattern and in Casanova we implement it by explicitly moving the moving object from a container of type `F` into an other container of type `T`. `T` and `F` share some information like physical information, graphics, etc. but differ in terms of behavior. We can generalize

the movement behavior by combining the above actions. We start with the definition of an entity `MovingActor` which is an entity that moves the position of its instance according to its velocity.

```
enity MovingActor =
        Resources : ResourcesName

        rule Resources.Position = yield Resources.Position +
                                        Resources.Velocity * dt
        //.. other rules and fields
```

An entity `ActionActor` is an entity that shares some structure with the `MovingActor` entity (for example the position or the velocity) but implements different rules.

```
enity ActionActor =
  Resources : ResourcesName

  rule Resources.Position = // move around a target for example
  rule Resources.Life = // remove life if the entity is hit
  //.. other rules and fields
```

A `SourceActor` is an entity that contains among its fields a `MovingActor` and an `ActionActor` field. `SourceActor` combines the actions described above so that when an entity of type `MovingActor` needs to behave like an `ActionActor` we use the deletion pattern to move the `MovingActor` into a temporary location, so as to give time to notify all the entities, and then we assign it to `ActionActor`. The code below shows this solution.

```
enity SourceActor =
  AActor : Option<ActionActor>
  ref AActorToDestroy : Option<ActionActor>
  MActor : Option<MovingActor>

  rule AActor, AActorToDestroy = // deletion logic code

  rule MActor =
    wait AActorToDestroy.IsSome
    yield Some(new MActor(AActorToDestroy.Value.Resources))
```

Next, we show how the just described Casanova 2 model effectively expresses an RTS. We do so by introducing a concrete case study and then its implementation in Casanova 2, which uses the just described model of an RTS game.

6.3 Implementation of a case study

We now implement a strategy game based on an already existing strategy game called Galaxy Wars game by means of the RAA pattern and the Casanova 2 language.

Galaxy wars (GW) is an RTS game published in 2012 inspired by the popular board game Risk. Galaxy wars has been used as a case study in related research [69]. The gameplay revolves around strategic choices, where timing, battles, and resource management are key elements to prevail against the opponents. The elements of Galaxy Wars that follow the RAA pattern are: *fleet*, *planet*, *statistic*, and *link*. Resources are *statistics*, the actors are *fleets*, *planets*, and *links*. The possible actions are movement, fight, and upgrade. In GW most of the entities are static. An actor that can be created

and deleted is fleet. A fleet is spawned after a player decides to send some units to a planet. A fleet is disposed after either it has reached its destination, or it has lost a battle. Moreover, the fleet actor is the only actor which might change its strategy/behavior during its lifetime (a fleet can either travel along a link or fight in a battle).

With the following code, which is complete, we wish to illustrate how little code is needed to implement such a game in Casanova 2, and how generic such code can be[2].

6.3.1 The world entity

We begin by defining the structure of the world entity. The world contains the collection of `Planets` in the map, the collection of `Links` connecting the planets, the collection of `Players`, and a `Controller` that manages the input controller and provides facilities like: the current selected planet, whether a mouse button is down, etc.

```
worldEntity GalaxyWars =
  Planets    : [Planet]
  Links      : [Link]
  Players    : [Player]
  Controller : Controller

  //rules
```

6.3.2 Resources

The resources are all those elements that influence the game dynamics. In Galaxy Wars the resources are:

- the players statistics (attack, defense, production, research)
- the planets statistics
- the fleets statistics
- the fleets stationed in a planet
- the fleets moving around the map

We use the properties below to model the statistics of the entities: player, planet, and fleet. We use these statistics to amplify or reduce the amount of resources to transfer, thus to alter the impact of the effects of the entity container.

```
entity GameStatistics =
  Attack               : float32
  Defence              : float32
  Production           : float32
  Research             : float32
```

[2]The complete working version of the this game can be found at `https://github.com/vs-team/casanova-mk2/blob/master/Unity/Tutorials/-GalaxyWars/Assets/World.cnv`, or can be requested to the author. Note that to run this game the Unity3D framework must be installed in the computer

6.3.3 Actors

Actors, or entities, represent the resource containers in Galaxy Wars. The entities in Galaxy Wars are:

- **Planet** which represents the container of stationed fleets. Each planet has its own statistics. Statistics affect: the attack and velocity of outgoing fleets, the local production of fleets, and the defense capabilities
- **Link** which is a directed connection between two planets. Links are used by fleets to move around the map
- **Fleet** which represents the armies of a player. A fleet is made up of ships, and statistics are assigned to every fleet. A fleet might behave differently depending on its current task. Therefore, we distinguish two different kinds of fleet, each of which inherits a base fleet and is able to accomplish a specific task. The kind of fleets that we identified are:
  - **AttackingFleet**, which is a fleet capable of carrying out fighting tasks;
  - **AttackingFleetToMerge**, which represents a special attacking fleet which has just conquered a planet and thus has to be added to the planet stationary fleets (together with the other allied attacking fleets who participated in the battle);
  - **TravelingFleet**, which represents a fleet traveling along a link;
  - **LandingFleet**, which represents a special traveling fleet which is about to land on the destination planet;
- **Battle** which carries out the fighting task on a planet
- **Player** which is the owner of entities in a game. Every player belongs to a faction. Factions differ from each other based on their statistics. During the game, the statistics of a player can be changed by means of upgrades

6.3.4 Fields

In addition to the resources defined above, additional data fields are used in every entity to support the internal logic of each entity. In what follows we go through each entity and for each entity list its fields.

Planet

Each planet has its own statistics, the number of stationed fleets, the incoming fleets, an owner, a link to a (possible battle), the landing fleets, an info about whether it is selected, an info about whether it has just been right-click selected, and its position.

```
entity Planet =
  Statistics    : GameStatistic
  LocalFleets   : int
  InboundFleets : [Fleet]
  ref Owner     : Option<Player>
  Battle        : Option<Battle>
  LandingFleets : [LandingFleet]
  Seleted       : bool
```

```
  RightSelected : bool
  Position      : Vector3
```

Link

Moreover, its source and destination, a link made up of a collection of traveling fleets.

```
entity Link =
  ref Source         : Planet
  ref Destination    : Planet
  TravellingFleets   : [TravellingFleet]
```

Fleet

A `Fleet` has statistics, the number of ships, a *ref* to the *link* on which it is traveling, an owner, a destroyed flag, and the position.

```
entity Fleet =
  Statistics  : GameStatistic
  Ships       : int
  ref Link    : Link
  ref Owner   : Player
  Destroyed   : bool
  Position    : Vector3
```

- **AttackingFleet** An attacking fleet is a specialized fleet that contains a *ref* to the actual fleet and a reference to its battle.

  ```
  entity AttackingFleet =
    ref MyFleet : Fleet
    ref MyBattle : Battle
  ```

- **AttackingFleetToMerge** An attacking fleet to merge is a specialized fleet that contains a *ref* to the actual fleet and a reference to the attacking fleet with which it has to join.

  ```
  entity AttackingFleetToMerge =
    ref MyFleet : Fleet
    ref FleetToMergeWith : AttackingFleet
  ```

- **TravelingFleet** Is a specialized fleet that contains a reference to the actual fleet, the destination planet, and the velocity.

  ```
  entity TravelingFleet =
    MyFleet : Fleet
    ref Destination : Planet
    Velocity : Vector3
  ```

- **LandingFleet** Contains the reference to the actual fleet.

  ```
  entity LandingFleet =
    MyFleet : Fleet
  ```

Battle

A battle is made up of the planet where the battle is taking place, a collection of attacking fleets, the losses of the hosting planet, the losses of the attacking fleets, the just destroyed attacking fleets, and the just arrived attacking fleets that have to be grouped into the attacking fleets.

```
entity Battle =
  ref MySource    : Planet
  AttackingFleets : [AttackingFleet]
  DefenceLost     : Option<int>
  AttackLost      : Option<int>
  FleetsToDestroyNextTurn  : [AttackingFleet]
  FleetsToMerge   : [AttackingFleetToMerge]
```

Player

A player is made of the statistics of its faction and his display name.

```
entity Player =
  Statistics : GameStatistic
  Name : string
```

6.3.5 Actions

Actions are the only way, according to RAA, to exchange resources like the amount of attacks in a battle, the number of fleets to produce, etc. In Galaxy Wars we identified three kind of actions: battle, production and upgrade.

Battle

A Battle action involves a planet `MySource` and a series of `AttackingFleets`.

- **Attack** In this design only one selected attacking fleet at a time can attack `MySource`, namely the fleet which is at the head of the `AttackingFleets` collection. Every few milliseconds damage is computed and stored in the `Battle` entity. Before computing the amount of damage, we check that there are still fleets in the `AttackingFleets` collection.

  ```
  entity Battle =
    ...
    rule AttackLost, DefenceLost =
      yield None, None
      wait 1.0f
      if AttackingFleets.Count > 0 then
        yield
          // amount of losses based on the
          // statistics of both the attacking
          // fleet and the planet
  ```

  The amount of damage represents the damage that has to be applied to both the selected attacking fleet and the defending planet. This damage will always be applied since every instance of `AttackingFleet` and `Planet` involved in a battle keeps updating the number of fleets.

```
entity AttackingFleet =
  ...
  rule MyFleet.Ships =
    wait MyBattle.AttackLost.IsSome &&
         MyBattle.AttackingFleets.Head = this
    yield MyFleet.Ships - MyBattle.AttackLost

entity Planet =
  ...
  rule LocalFleets =
    wait Battle.IsSome && Battle.DefenceLost.IsSome
    yield LocalFleets - Battle.DefenceLost.Value
```

- **Attacking fleet selection** A random selection is used to allow all attacking fleets to attack the planet.

```
entity Battle =
  ...
  rule AttackingFleets =
    .| AttackingFleets.Count <= 1 => yield AttackingFleets
    .| _ =>
      wait Random.Range(1.0f, 2.0f)
      yield AttackingFleets.Tail @ [AttackingFleets.Head]
```

- **Ownership** We change the owner of a planet when at the end of a battle the attacker list is not empty. When we change the owner we also update the number of `LocalFleets`, by adding all the fleets that share the same new owner and that are attacking the planet.

```
entity Planet =
  ...
  rule Owner, LocalFleets =
    if Battle.IsSome &&
       LocalFleets = 0 &&
       Battle.AttackingFleets.Count > 0 then
      let new_owner = Battle.AttackingFleets.Head.MyFleet.Owner
      let fleets_to_add =
        Battle.AttackingFleets
        .Where(f => f.MyFleet.Owner = new_owner &&
                    f.MyFleet.Ships > 0)
        .sum(f => f.MyFleet.Ships)
      yield Some new_owner, fleets_to_add
```

Production

The spawning of a new fleet follows a simple schema: if a battle is ongoing on a planet then production is interrupted and the planet keeps polling the battle in order to update its local fleets; if the planet is neutral (it is not possessed by any player) then production does not take place; eventually if the planet is not neutral and there is no ongoing battle then we wait some time, which depends on the production statistics of both the player and the planet, and then we add a new fleet to the local fleets.

```
entity Planet =
  ...
  rule Owner, LocalFleets =
    .| Battle.IsSome => yield LocalFleets
    .| Owner.IsNone => yield 0
    .| _ =>
      wait T //time depending on the owner statistics
      //and the planet production statistics
      yield LocalFleets + 1
```

Upgrade

When the planet is selected and a key associated to an upgrade is pressed, we: (*i*) wait some time (depending on various stats), and then (*ii*) we upgrade the selected statistic. If the planet is neutral then its statistics are kept to 1.

```
entity Planet =
  ...
  rule Statistics.STAT =
    .| Owner.IsNone -> yield 1
    .| _ ->
      wait IsSelected && KeyPressed(STAT_KEY)
      wait //time depending on the owner
      //and the planet research
      yield max(MAX_STAT, Statistics.STAT + 1)
```

6.3.6 Creation

In Galaxy Wars we create entities when: (*i*) a battle is about to start, and (*ii*) when a fleet is spawned.

Battle

On a planet a battle is created either when the planet is neutral and a fleet is approaching the planet; or the planet is not neutral, there are no battles ongoing on the planet, and an enemy fleet is approaching the planet.

```
entity Planet =
  ...
  rule Battle =
    let exits_an_enemy_fleet =
      LandingFleets.Count = InboundFleets.Count |> not
    if (Owner.IsNone && Battle.IsNone && exits_an_enemy_fleet) ||
       (Owner.IsSome && exits_an_enemy_fleet) then
      yield Some (new Battle(this))
      wait Battle.AttackingFleets.Count <= 0
      yield None
```

Fleet

We consume all local fleets of a planet and move them through the link when the source planet is selected (and its fleets are greater than 0) and the destination planet is selected as well.

```
entity Link =
  ...
  rule TravellingFleets, Source.LocalFleets =
    wait Source.Selected && Destination.RightSelected &&
    Source.Owner.IsSome && Source.Battle.IsNone &&
    Source.LocalFleets > 0
    yield new TravellingFleet(Destination) :: TravellingFleets, 0
```

In the following the selection logic of a planet is presented. Note the function `IsMouseOver`, which might vary depending on the adopted rendering framework, for example, when using Unity3D `IsMouseOver` becomes a property, which returns a data of type `bool`, and is exposed by the proxy attached to the planet. `IsMouseOver` in this case would be internally managed and updated by the proxy itself.

```
entity Planet =
  ...
  rule Selected =
    wait Input.GetMouseButtonDown(0) &&
         not (Input.GetKey(KeyCode.LeftShift) ||
         Input.GetKey(KeyCode.LeftControl))
    yield IsMouseOver(Position)

  rule RightSelected =
    wait Input.GetMouseButtonDown(0) &&
         (Input.GetKey(KeyCode.LeftShift) ||
          Input.GetKey(KeyCode.LeftControl))
    if IsMouseOver(Position) then
      yield true
      yield false
```

6.3.7 Deletion

Analogously to creation, in Galaxy Wars the entities which might be disposed during a game are battles and fleets.

Battle

The logic of the deletion of a battle is tightly related to the logic of its creation. In the previous subsection a battle is disposed only when the amount of `AttackingFleets` is equal to `0`.

Fleet

The general logic of deletion of a fleet is as follows: if the fleet has no ships then it has to be destroyed. In code, a fleet destroys itself when the number of its `Ships` is less or equal to zero.

```
entity Fleet =
  ...
  rule Destroyed =
    wait Ships <= 0
    yield true
```

Fleets can be specialized for fighting, traveling, or landing during their lifetime.

- **Fighting** If during a battle the attacker manages to conquer the planet then all the attacking fleets that share the same owner of the just conquered planet have to be destroyed.

```
entity AttackingFleet =
  ...
  rule MyFleet.Destroyed =
    wait (MyBattle.MySource.Owner.IsSome &&
          MyFleet.Owner = MyBattle.MySource.Owner)
    yield true
```

They are then filtered from the attacking fleets collection and moved into `FleetsToDestroyNextTurn`. We move such *fleets to destroy* in a different collection so their logic will not affect the logic of the battle. Fleets to be destroyed stay in the list exactly one frame.

```
entity Battle =
  ...
  rule FleetsToDestroyNextTurn =
    yield
      [for f in AttackingFleets do
       where (MySource.Owner.IsSome &&
       f.MyFleet.Owner = MySource.Owner))
       select f]
```

Fleets that have not managed to conquer the planet, but which have been destroyed, are filtered from the attacking list collection.

```
entity Battle =
  ...
  rule AttackingFleets =
    yield
     [for f in AttackingFleets do
      where (not f.MyFleet.Destroyed) &&
      not FleetsToDestroyNextTurn.Contains(f)
      select f]
```

- **Landing** A landing fleet is a travelling fleet which is about to land and which owner is the same as its destination planet. In order to not add twice the ships of the landing fleet to the local fleets of the destination planet, a landing fleet stays exactly one frame in the game.

```
entity LandingFleet
  ...
  rule MyFleet.Destroyed = yield true
```

New landing fleets are continuously added to the local fleets.

```
entity Planet
  ...
  rule LocalFleets =
    yield LandingFleets.Sum(f => f.MyFleet.Ships) +
          LocalFleets
```

- **Traveling** When a traveling fleet has reached its destination, the fleet is automatically filtered by the link.

```
entity Link =
  ...
  rule TravellingFleets =
    yield
      [for f in TravellingFleets do
       where (f.MyFleet.Destroyed |> not &&
              Vector3.Distance(f.MyFleet.Position,
                               Destination.Position) >
              Destination.MinApproachingDist)
       select f]
```

6.3.8 Strategy update

An entity during its life cycle might change its behavior based on its state. An example of this kind of behavior in Galaxy Wars could be identified in the fleet entity. For example an attacking fleet behaves different from a moving fleet. In Casanova 2 we distinguish these two cases by means of two different entities that share some common properties, but implement different rules.

Inbound Fleets

When a fleet, travelling along a link, is approaching its destination, the planet has to choose whether to: (*i*) add the fleet to the planet's local fleets (see production of a planet above), or (*ii*) add the fleet to a battle. To implement the just described scenario we start with the definition of a buffer to place in the `Planet` entity called `InboundFleets`. The `InboundFleets` of a planet represents all fleets that are approaching the planet.

```
entity Planet =
  ...
  rule InboundFleets =
    yield [for l in world.Links do
           where (l.Destination = this)
           for f in l.TravellingFleets do
           where (Vector3.Distance(f.MyFleet.Position, Position) <=
                  MinApproachingDist)
           select f.MyFleet]
```

`InboundFleets` acts like a dispatcher. When a fleet enters the `InboundFleets` collection, other entities are able to consume it for their internal logic. To avoid entities to consume twice the same fleet, fleets in `InboundFleets` last for one frame before being disposed. When an entity consumes an

inbound fleet it decides what behaviors to apply to the selected fleet. This is done by assigning the fleet to an other instance, of different type, which contains the fleet but provides new rules.

Attacking fleets to merge

Fleets that come from the same link and that share the same owner have to be joined. To do so, every enemy inbound fleet that shares the same source link of a fleet stored in `AttackingFleets` is selected and converted into an `AttackingFleetToMerge`. Eventually all the fleets of type `AttackingFleetToMerge` are stored into `FleetsToMerge` for one frame.

```
entity Battle =
  ...
  rule FleetsToMerge =
    yield
      [for i_f in MySource.InboundFleets do
       for a_f in AttackingFleets do
       where (not a_f.MyFleet.Destroyed &&
              i_f.Link = a_f.MyFleet.Link)
       select (new AttackingFleetToMerge (i_f, a_f))]
```

When we create an attacking fleet to merge the reference to the actual attacking fleet is stored in the `FleetToMergeWith` of the attacking fleet to merge. An attacking fleet iterates every frame all the attacking fleets of its battle and selects those fleets to merge whose `FleetToMergeWith` is the attacking fleet in question. After selection, the number of ships of the selected attacking fleets to merge is added to the local fleets of the attacking fleet.

```
entity AttackingFleet =
  ...
  rule MyFleet.Ships =
    yield MyBattle.FleetsToMerge
          .Where(f => f.FleetToMergeWith = this)
          .sum(f.MyFleet.Ships)
          + MyFleet.Ships
```

Attacking fleet

A battle entity selects the enemy fleets from the inbound fleets of its `MySource` field and adds them to its `AttackingFleets` every frame, as long as the selected fleets are not in `FleetsToMerge`. Before adding the inbounding attacking fleets, every inbound enemy fleet is converted to an attacking fleet.

```
entity Battle =
  ...
  rule AttackingFleets =
    yield
      [for f in MySource.InboundFleets do
      let is_ship_to_merge = FleetsToMerge.Contains(f)
      where is_ship_to_merge &&
            (MySource.Owner.IsNone ||
             not (f.Owner = MySource.Owner))
      select new AttackingFleet(f, this)]
```

The moment a fleet becomes an attacking fleet and is added to the `AttackingFleet` collection the new attack logic can be run.

Landing fleet

Finally, A planet selects all allied fleets among its inbound fleets and adds them to the `LandingFleet` collection every frame, so the planet can later add those fleets to its local fleets. Before adding the inbound allied fleets, every fleet is converted to an inbound fleet.

```
entity Planet
  ...
  rule LandingFleets =
    if Owner.IsSome then
      yield
        [for inbound_fleet in InboundFleets do
         where (inbound_fleet.Owner = Owner)
         select (new LandingFleet(inbound_fleet))]
    else yield []
```

6.4 Summary

In this section we showed the idioms of the Casanova 2 language, which help developers with designing and building games in practice. Moreover, we presented a detailed and extensive example of the implementation of a game in Casanova 2. This game has been used to show not only the implementation details necessary to implement a video game in Casanova 2, but also to give the reader the idea of how complex behaviors are encoded in Casanova 2, in an idiomatic fashion.

Chapter 7

Evaluation

This chapter discusses the evaluation we made of Casanova 2 to ensure that it fulfills the requirements defined in Section 2.3. We present two different kinds of evaluation, one analytical based on the observation of some objective properties which are not directly measurable, and one quantitative based on direct measurements. We have chosen these two kinds of evaluation, to measure the attributes of the Casanova 2 language from different perspectives: (*i*) the analytical evaluation discusses features of Casanova 2 related to the *experience* of using it in practice; whereas (*ii*) the quantitative evaluation discusses the attributes of Casanova 2 with respect to other representative languages used for game development, such as performance and amount of code needed to encode game aspects.

7.1 Analytical evaluation

In Section 2.4 we discussed the advantages that result from the adoption of doman specific languages (DSL's) for developing video games. However, the discussion was not centered around Casanova 2. In the following we get back to those advantages, but this time we present them with specific focus on Casanova 2. In the following subsections we discuss how the features provided by the Casanova 2 language realize the advantages introduced in Section 2.4. Specifically, the features that that we will discuss are: writing, readability, optimization/performance, interoperability, and genericity.

7.1.1 Ease of writing

A fundamental requirement for a DSL is that it allows the definition of programs belonging to its domain. Being a DSL, Casanova 2 comes with a series of domain abstractions that cover typical aspects of game development such as the flow of time, suspensions, and rules. Among these abstractions we find constructs such as `wait`, `rule`, `world`, `dt`, etc. When compared to other tools for game development, these Casanova 2 abstractions offer a clear advantage in terms of development time and compactness. This is due to the fact that Casanova 2 abstractions capture complex behaviors of games, which by means of a GPL would require considerable time to be expressed and tested, as most of the constructs of a general purpose language (GPL) are generic and for general usage. For example, to implement a rule the body of which is suspendable in Casanova 2 a developer would write the rule body in an imperative fashion, but with the possibility to use keywords such as `wait` that allow the suspension of statements. In contrast, to simulate a suspendable rule in a GPL, a series of switches, each expressing the various nested steps, would be necessary. Suspending a rule by means of these switches would require the developer to set up by hand how and where to resume the next iteration. Some GPLs come with higher level constructs and helpers for concurrent behavior, such as coroutines. However,

even in this case additional code that coordinates such mechanisms is necessary. All this code makes the program harder to maintain, since lots of spurious aspects must be encoded which are not directly related to the game design, but are still necessary to maintain and emulate temporal behaviors.

This lack of noise in the code is visible in different games written in Casanova 2, examples of which we find in Chapter 6 and in Appendix B. The measurements allow us to conclude that Casanova 2 offers evident advantages in terms of development time and compactness of the resulting game code.

7.1.2 Readability

Being easy to write is not a sufficient requirement for a DSL, since the development of a game is a dynamic process made up of several phases in which, before achieving the final goal, the code might change. During these phases it is important for the code to be readable, in order to be maintainable. In order to be readable the DSL itself must provide constructs, in the shape of domain abstractions, that facilitate this property, which help developers understand their own code and to map it to the design elements present in the game.

Indeed, capturing typical aspects of video games with specific language abstractions, positively affects compactness: by means of one abstraction we can express many concrete and complex behaviors. Casanova 2 captures, by means of domain specific abstractions, the typical aspects of games such as the flow of time, suspensions, and rules.

As a game requires less and less unrelated considerations hard-coded in the game code, the final game code becomes more readable and maintainable. This is due to the fact that when dealing with game code developers will find more and more considerations (in the form of Casanova 2 instructions) that are directly mapped to elements present in the game design.

This property is also backed up by the quantitative evaluation carried our in Section 7.2.3. Moreover, a preliminary experiment to measure the readability of Casanova 2 code has been carried out during a workshop session on Casanova 2. A description of the workshop is summarized in Appendix A. Results of this preliminary work show that Casanova 2 code is readable by senior developers who never developed games using Casanova 2 before.

In conclusion, Casanova 2 code is highly readable, and therefore maintainable. As the quantitative evaluation in Section 7.2.3 will show, this result holds even when compared to other tools for game development.

7.1.3 Optimizations/Performance

A DSL for games that is readable, and allows the definition of compact programs, should also hide considerations that are not part of the game design, since such considerations might affect the readability of the code. In games it is often the case that developers have to consider in their code non-functional requirements, such as performance, which are not part of the game design itself. Therefore, a DSL for games should be able to capture such requirements without the direct intervention of developers, thus without affecting the readability of the original input code.

Due to the specificity of the application domain and of the provided Casanova 2 abstractions, the Casanova 2 compiler (as discussed in Chapter 4 and 5) is able to effectively use code analysis techniques and to apply them to generate code that exhibits fast runtime execution, without the direct intervention of the developer to encode faster algorithms by hand. For example, by using statements that suspend the execution of blocks of code, such as `wait` or `yield`, the Casanova 2 compiler is able to identify those parts of the code that need to be suspended, and to use this information to generate a more efficient (yet equivalent) version.

Another example is provided by the encapsulation optimization discussed in Chapter 5. This optimization allows developers to write programs that exhibit high encapsulation without losing per-

formance at runtime, because of the many messages passed between the different, encapsulated, interacting entities in a program. This is possible because the Casanova 2 compiler transforms the input program into an equivalent version where a source entity updates directly a target entity, instead of asking the target entity to update itself with new data from the source entity. As this reduces the number of communications, the overall runtime performance is improved.

This property has been measured with two scenarios where we compared the performance of Casanova 2 games against the performance of equivalent implementations, but written with other representative languages used in game development.

The results of these experiments, which are found in Section 7.2.2, show that Casanova 2 supports the making of games that exhibit high-performance at runtime.

7.1.4 Interoperability

Being a DSL does not guarantee the fact that Casanova 2 can tackle all the aspects of game development, such as rendering, content management, etc. Casanova 2 should support interoperability in order to be able to delegate such tasks, which are not integrated in Casanova 2, to existing tools.

Casanova 2 allows interoperability due to the architecture of its compiler. Specifically, Casanova 2 offers a layered compiler that allows developers to extend it whenever needed. For example, the last layer transforms the intermediate code, generated during the various compilation stages, into an equivalent version but written in a different language; if a developer needs to target new engines or platforms, he/she will only need to adapt the last layer to include the rules of how to translate the intermediate code into the target language.

The Casanova 2 compiler also allows Casanova 2 programs to interface with other game engines or frameworks without the necessity of a predefined layer in the compiler that regulates the interaction. This can happen in two different ways: either by referencing the library directly in the Casanova 2 game code, or by means of the so-called proxy system. The proxy system acts as an adapter between the targeted framework and the Casanova 2 code.

To support this property, and more specifically to show that Casanova 2 can interoperate with external libraries or frameworks, in the following we show different examples of frameworks that interoperate with Casanova 2 code. The choice of these frameworks is derived from the fact that we want to show how even if the frameworks implement different architectures, we can still connect Casanova 2 code to them, since Casanova 2 is framework independent.

In order to show that Casanova 2 achieves interoperability, in the following we show three applications of Casanova 2 with different frameworks: MonoGame, Unity3D, and Lego mindstorms V3. For all the three examples we used the proxy system to interface Casanova 2 code with the external framework.

MonoGame

MonoGame is a low-level framework for general game development, which is nowadays used by many developers. MonoGame is an open source framework based on XNA (another widely used tool for game development, which has been discontinued). It is mainly used for 2D games, and was the first engine supported in Casanova 2. Among the first applications made with MonoGame and Casanova 2, we find the first version of the Dyslexia game (Figure B.3a), used as research tool at Tilburg University

As MonoGame comes with its own game loop, the Casanova 2 compiler processes Casanova 2 code and generates C# code that is later included in a MonoGame project. Once included, the Casanova 2 game is then run inside the MonoGame framework. Casanova 2 code then interacts with MonoGame facilities through the proxy system (see Section 4.3).

Unity3D

Dealing only with low-level game engines such as MonoGame by itself does not guarantee that Casanova 2 is also suitable for interoperating with more complex and higher-level engines. Therefore, to assess that Casanova 2 is suitable for interoperating with complex and high-level game engines, we tested Casanova 2 together with the Unity3D engine.

We chose the Unity3D engine, since it is widely used by the game development community, and is becoming a standard for many game development contexts. Among the examples of Casanova 2 and Unity3D games we find all the games discussed in Appendix B, and those implemented internally within our research team (see Figure 7.1). The examples in Figure 7.1 are demonstrations of simulations implemented in Unity3D for demo purposes. Specifically, the demo in Figure 7.1a shows how to simulate complex physics in Casanova 2 through an asteroids field simulation, whereas the demos in Figures 7.1b and 7.1c show how to handle the user's input in Casanova 2, and eventually the demo in Figure 7.1d shows an AI controlling a patrol's behavior in Casanova 2.

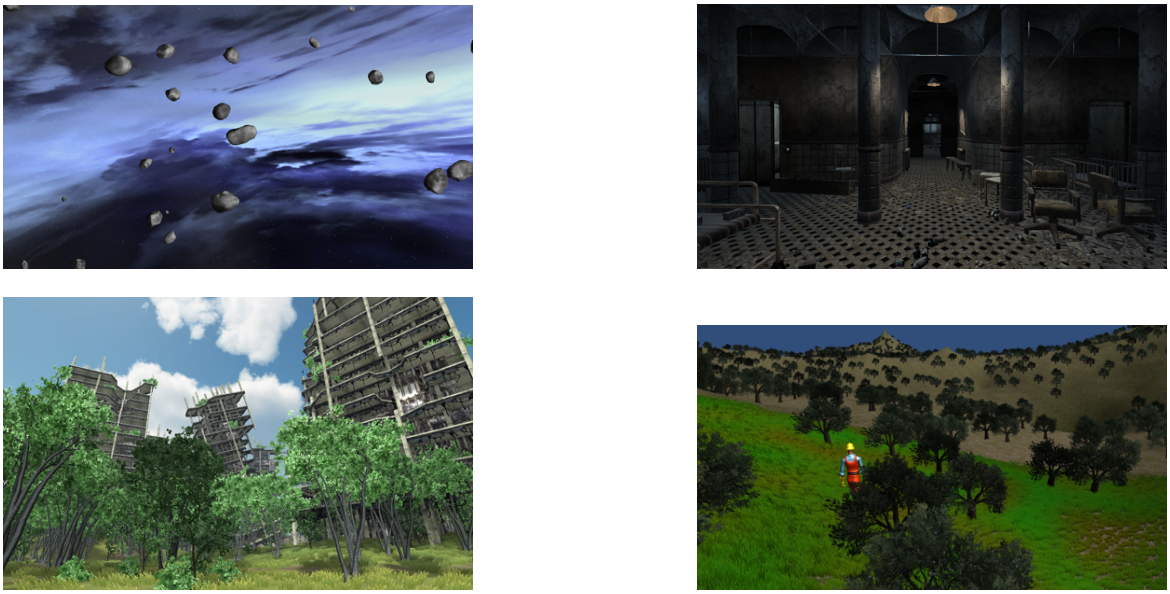


Figure 7.1: Some Casanova 2 games internally produced

As Unity3D comes with its own game engine, the Casanova 2 compiler processes Casanova 2 code and generates C# code that is then run by the Unity3D engine. Casanova 2 code interacts with Unity3D through the proxy system, similarly to MonoGame.

Lego mindstorms V3

We also explored usages of Casanova 2 with other frameworks that are not meant for game development such as the Lego mindstorms V3. Lego mindstorms V3 is not a game engine, but it exhibits similarities to the game development field, such as concurrency, and manipulation of behaviors depending on time. Lego mindstorms is a kit that contains software and hardware meant to create and customize robots. We successfully managed to run a Casanova 2 program to steer a robot (see Figure 7.2).

As Lego mindstorms is only meant to be controlled remotely, the Casanova 2 code is run inside a local game loop built ad-hoc by the Casanova 2 compiler. Casanova 2 code interacts with the remote

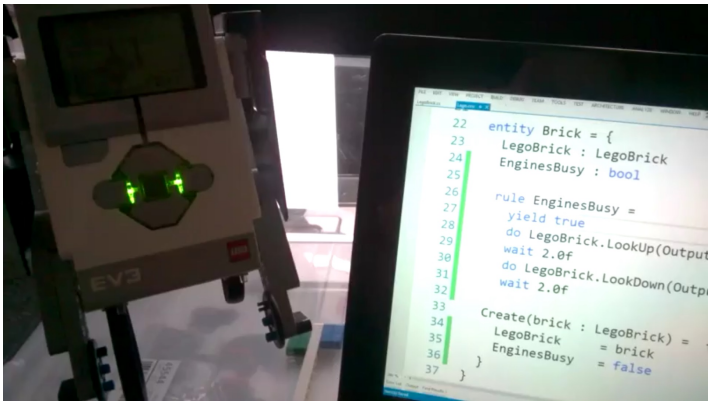


Figure 7.2: A Casanova 2 running a Lego mindstorms V3 program

elements of the robot through the proxy system.

From these experiences we conclude that Casanova 2 is suitable for interoperating with a high variety of front-ends, whether it is low-level (MonoGame), high-level (Unity3D), or outside the realm of games altogether (Mindstorms).

7.1.5 Genericity

Interoperability by itself does not guarantee that we can actually develop any possible video game. It is up to the Casanova 2 language to ensure the possibility to generically build all sorts of games. Indeed, the building blocks of Casanova 2, i.e., its syntax and semantics, are designed to support the definition of games regardless of their genre or structure. The genericity of these building blocks has been assessed by a series of games made with the Casanova 2 language. In the following we list some of these games:

- *Galaxy wars*, a real time strategy game
- *Zombie shooter*, an "escape the city", multiplayer shooter game
- *Dyslexia*, a game for detecting dyslexia in children
- *3D asteroid shooter*, a 3D asteroids shooter game
- *Contact*, a multiplayer game for studying the evolution of language

It is worth noticing that most of the games, and samples, made with the Casanova 2 language were developed by bachelor students of computer science, who learned Casanova 2 specifically to produce these games. A detailed explanation of these games and a discussion of the developers and their background can be found in Appendix B.

On junior game developers During the year 2016 we tested Casanova 2 usability by asking junior developer to build games with it. These tests, besides showing us that it is possible to implement different kinds of games, showed us that the language syntax is mature enough to allow junior developers with little knowledge of the language to successfully build games. Examples of the games built and a description of the developers' background can be found in Appendix B.

On senior game developers Moreover, we also made an investigation on the perceived quality of Casanova 2 by expert game developers. This experiment, which took the shape of a workshop at the University of Amsterdam, showed us that while Casanova 2 is missing some important technological aspects, such as a real-time debugger, it is still usable by expert developers. Casanova 2 is appreciated by expert developers, and the perceived advantages of the language overlap with what we expect to be the strong points of Casanova 2. More details regarding this workshop, and the questionnaire we gave to the participants, can be found in the Appendix A. Due to the relatively low number of participants at the workshop, it does not make sense to statistically analyze their answers to the questionnaire we provided, though we can still convey our impression of their responses.

7.2 Quantitative analysis

In this section we discuss the quantitative analyses of attributes of Casanova 2 which are important for game development. We will present such attributes by comparing them against other representative languages used for game development. The attributes that we will discuss in the following section are performance and code length. These two attributes are important since a game must run fast and at the same time its code should be compact and readable in order to remain maintainable.

In section 7.2.1, we discuss the set up and criteria used to evaluate Casanova 2. Sections 7.2.2 and 7.2.3 report on the evaluation of the program's performance and compactness/readability respectively.

7.2.1 Set up and goal of the evaluation scenarios

In evaluating Casanova 2, we consider performance and readability as most crucial evaluation criteria.

Performance Performance is a fundamental indicator of the feasibility of a programming language that needs to be used in a resource-conscious scenario such as games, since every feature in a game comes with a series of costs in terms of CPU cycles. As a game starts including more and more features, the demands on the CPU increase as well. As the game grows more complex, eventually the CPU stops being sufficiently powerful to render one frame in the time needed for a smooth experience. Games are real-time applications, thus high-performance is required in order to keep the overall game experience immersive. When high performance is affected negatively, for example, because of the presence of complex features, the developers have to choose between either removing the involved features (or reduce them in effect), or re-factoring the involved features in order to (if possible) achieve higher-performance. The first option has an evident negative effect on the game design, since fewer initial design considerations will appear in the final version of the game. The second option has as problem that, due to the fact that it is accompanied by an increase in code size and complexity, maintainability of the code is affected negatively.

Readability Every feature in a game needs to be written by means of some code. As the number of features increases the code to express them increases as well. When a language is not suitable to "naturally" express some features, the amount of code necessary to express them becomes very large, since every "hard to express" feature (and each of its sub-aspects) will require lots of awkward, verbose code. As an immediate effect of such increase in code size is that game code becomes less readable. When readability is affected negatively as an immediate effect we have that maintainability is affected negatively as well. This has an cascading effect on other important aspects in game development, such as debugging and adding new features to the game, since they now will become more complex and will take more time to implement. Thus, readability of the code of a game is an important attribute that has to be watched closely in order to preserve the feasibility of the game development process.

Ideally what we would like to achieve is a game of which the code is not only maintainable and compact, but also of high performance. The goal of this quantitative evaluation is to find whether Casanova 2 allows the definition of programs that exhibit high performance, and which code is at the same time readable and concise.

Tested languages We have chosen four comparison languages which represent various development styles and which are all used in practice for building games. We have chosen a variety of languages which all exhibit various mixtures of performance and succinctness, with the goal of testing Casanova 2 as a language that captures the useful attributes of these languages in game development. We have mostly focused on those languages which are used for building game logic, and we have shied away from considering languages such as C++, which are used for building engines or libraries [50], as Casanova 2 is not in direct competition with them and therefore a comparison would be meaningless. Three of the chosen languages are dynamically typed programming languages: Lua, JavaScript, and Python, which have as their main selling points simplicity and immediacy [52]. The fourth chosen language is C# because of its good performance and relative simplicity when compared to C++.

The four languages mentioned above are all used when discussing the general performance and code length of Casanova 2 programs (scenario 1). However, C# is the only language used in the second part of the evaluation (scenario 2), since it is the only language (among the ones described above) that, according to the first part of this evaluation, has a performance profile comparable to that of Casanova 2.

Set up We tested the two evaluation variables of Casanova 2, performance and readability, using two different scenarios. In Section 7.2.3 we discuss the performance results in each of these scenarios, in Section 7.2.2 the readability results. For the two evaluation variables, we compare Casanova 2 programs with equivalent ones written in other languages.

Scenario 1 simulates a scenario featuring ten thousands patrols moving in different directions. The number of patrols is large in order to simulate the crowded scenes of a typical video game. The choice of the patrol is derived from the fact that patrols offer discrete dynamics, which we often find in games, such as waiting until the next checkpoint is reached. Thus, the combination of thousands of entities all performing complex, intertwined, and nested behaviors offer an interesting benchmark scenario from both perspectives: *performance*, since a high amount of entities featuring complex interactions stress the CPU , and *code length*, since defining complex operations that exhibit nesting behavior can require lots of code which amount can increase dramatically when the support of the chosen tool is limited.

Scenario 2 simulates a generic game in which one thousand entities are spawned every 5 seconds (initially the game starts with 10000 entities). When spawned every entity stays inactive for a random amount of time (between 5 and 10 seconds) before getting activated. When activated, the entity starts moving for a random period of time (between 4 and 8 seconds), and eventually when this time is elapsed the entity gets destroyed.

This simulation is built to get a systematic evaluation of the proposed approach to the encapsulation optimization discussed in Chapter 4 for both performance and code length: *performance*, since a scenario is built to test the performance of the fast wakeup collection[1] (entities stay inactive for a few seconds before getting activated) against a continuously polling solution that checks the states changes, and *code length*, since building such optimization by hand requires a considerable amount of

[1]For this evaluation additional conditions are added (with different timers) to each entity, in order to make the simulation dynamics more articulated and "heavy" in terms of amount of code to run.

Table 7.1: Performance comparison

Language	Time per frame
Casanova 2	0.07ms
C#	0.12ms
JavaScript	24.07ms
Lua	20.90ms
Python	20.15ms

code, which we do not find in the game description, that with Casanova 2 comes "for-free" due to the compiler analysis.

7.2.2 Performance

Performance is a fundamental indicator of the feasibility of a programming language that needs to be used in a resource-conscious scenario such as games. In particular, we observe that, in many cases, programming languages for games offer a difficult either-or choice between simplicity and performance. In the following, we show how Casanova 2 code can achieve high runtime performance, by means of a series of evaluations on resource-consuming game scenarios. We will use the outcome of these evaluations to compare the performance of Casanova 2 code against the performance of other representative languages used in game development.

More specifically, in this section we discuss the performance of Casanova 2 by means of two different scenarios. In the first one we show the performance of Casanova 2 and its constructs against other representative languages used in game development, which we used idiomatically. This evaluation shows how by translating the bodies of Casanova 2 rules into the flat state machines introduced in Section 4.2.4 we can achieve high-performance at run-time. The second evaluation presents the performance gained at runtime by Casanova 2 games when switching on, in the compiler, the optimization introduced in Section 5.

Scenario 1

In this scenario we made an effort towards implementing the sample by using coroutines and generators [71] whenever available, in order to express the game logic in an idiomatic style for each of the tested languages. In order to compare the language functionality, we are only running the logic of the game and we do not execute any other unrelated components, such as the graphics engine, which might otherwise pollute the outcome of our evaluation. The code samples can be found on [1]. For this first scenario we used Casanova 2 with the compiler introduced in Chapter 4, without the extra optimizations that were introduced in Chapter 5.

Results We have generated tens of thousands of entities in a loop that simulated a hundred thousand frames. This corresponds roughly to half an hour of play time on a reasonably crowded scene. The results are summarized in Table 7.1.

As we can see from the table, the performance of Casanova 2 is of the same order as C#, and is multiple orders of magnitude faster than that of the scripting languages. In this simple but populated scenario, the limits of Lua, Python, and JavaScript, deriving from the high cost of dynamic lookup, are clearly shown. For example in a statement such as `s.p.x.a.b()` there are 4 lookups, each of which will first check the existence of the attribute, and then proceed to actually dereference the attribute. In addition, all languages idiomatically implement state machines by means of coroutines

and generators. The techniques abundantly use virtual calls, which add overhead at the expense of performance.

The big advantage of Casanova 2 is that it uses all the static information available, in order to avoid work at runtime while still offering expressive and concise constructs. For this reason, code such as `yield a, b, c` translates directly into assignments, while retaining higher level semantics.

In Figure 7.3 a detailed comparison between the code of the Casanova 2 program and the equivalent Python implementation is provided. We chose only Python for a detailed comparison, because as all three dynamics languages roughly exhibit similar verbosity and performance. Furthermore Python shares profiling tools with C# and Casanova 2 via Visual Studio, therefore a closer performance evaluation is possible.

Every node of the flow graph in Figure 7.3 represents a computational element of the running example (for example `Patrol Update` represents the code that takes care of calling the rules, or methods in case of Python, that modify the velocity and the position of the patrol). Moreover, every node comes with two tables each representing the performance of that specific computational element in Casanova 2 and Python respectively. Every table comes with a series of records each representing various aspects such as total execution time in seconds, and total percentage of time spent in this node. Note that each item comes with an inclusive (incl) and exclusive (excl) measure: inclusive means that its amount includes not only the time spent in the current node, but also the time spent in all the children of this node; exclusive means that its amount does not include the time spent in the children of this node, but only the time spent in the current node. For example in the node `Update Patrol` consider the CNV table: 18 ms excl represents the total time that the program spent in this node; 7.83% excluded (excl) represents the total percentage time the program spent in this node; and 53.58% incl represents the total percentage time the program spent in this node and in its children.

We notice that Casanova 2 is in general faster than Python, due to the fact that Casanova 2 is compiled whereas Python is interpreted. Thus, optimization techniques, such as resolving call addresses at compile time, or inlining instructions, typical of statically compiled languages, do not apply to interpreted languages, such as Python. This translates into code that is generally less performant. Moreover, by adopting flat state machines to capture interruptible code blocks, note this is possible only because of our code analysis done at compile time, we avoid the typical overhead deriving from using design patterns (such as the strategy pattern) for capturing abstract behavior (such as coroutines). Indeed, in Python, to suspend and later resume a piece of code, a supporting data structure is necessary in the form of a coroutine to emulate this behavior. However, this comes at a cost, since lots of virtual calls and dynamic dispatching are necessary in order to maintain the coroutine generic with respect to a concrete implementation. We can see in Figure 7.3 the cost of using coroutines in Python by looking at the node called `Next` (at the bottom of the figure). This node takes care of moving the iteration from one coroutine state to another. As we can see this computational element consumes about 30% of the total spent time just for switching between one state of the program and another, whereas in Casanova 2 changing from one state to another requires the program to just set one single integer variable: the next state.

However, being statically compiled does not provide such an obvious advantage since languages, such as C# still run slow, because of their generality and incapacity of applying domain specific optimization, such as the one proposed in Section 4.2.4. Indeed, just like in Python, coroutines in C# use the same philosophy (both are based on the strategy pattern), which translates into additional resources to maintain and update the data structures necessary to support the coroutine and therefore allow blocks of code to suspend.

In the following we discuss the second scenario, which evaluates the performance of the optimization over encapsulated code introduced in Chapter 4.

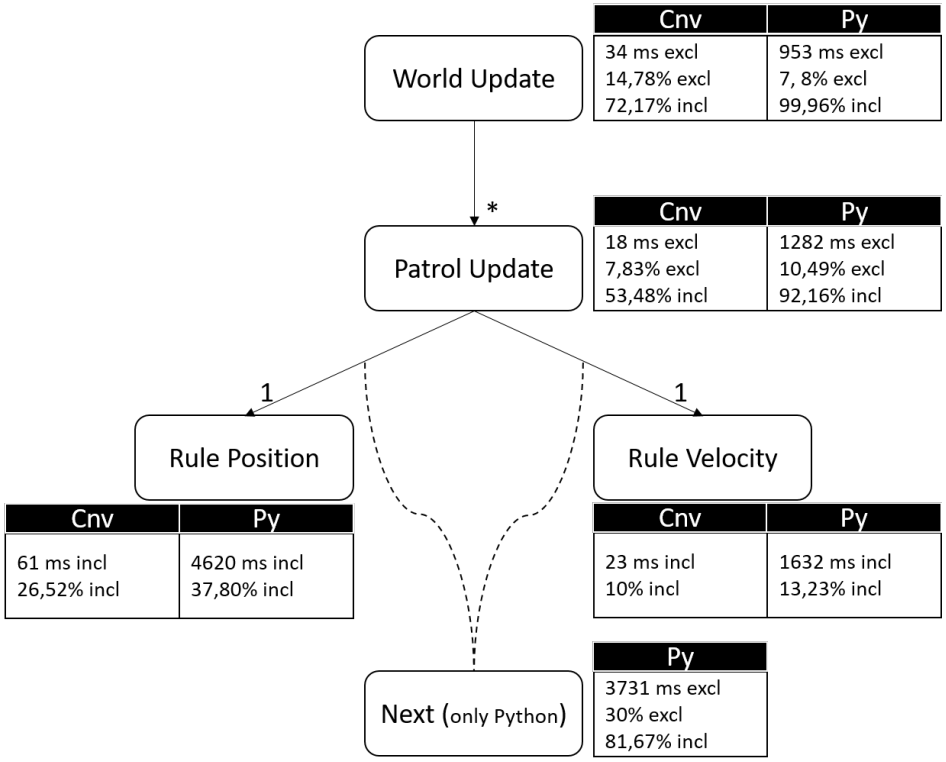


Figure 7.3: Performance comparison between Casanova 2 (Cnv) and Python (Py). In the figure, for each operation/processing module, the runtime is given (in ms), and the percentages of total execution time spent in it, or in it and in its children, expressed as excl and incl, respectively.

Scenario 2

In the second scenario, we discuss the performance of two compilation outputs of Casanova 2 against an equivalent implementation in the C# language. The first compilation output is generated by the standard Casanova 2 compiler, whereas the second compilation output is generated by an alternative version of the Casanova 2 compiler, which implements the optimization technique introduced in Chapter 4. Moreover, in this evaluation we also discuss the performance of the two different generated outputs against an idiomatic, but equivalent, implementation in the C# language. We also run the code with two different game engines, namely Unity3D and MonoGame, both using .Net but of different versions. The reason of this choice is derived from the fact that we want to show the generality of our optimization, no matter the concrete implementation or framework running it.

Results For each output we measure the time (in milliseconds) that it takes to fully complete the logic of a single frame (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 and add unwanted noise.

Table 7.2 shows the performance results of our scenario. As we can see for both frameworks (Unity3D and Monogame) the performance of our optimized Casanova 2 code is higher than the one running the non-optimized version 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 game loop in the underlying frameworks.

Table 7.2: Running time comparison

Platform	Language	Optimized	Performance
Monogame	Casanova 2	No	0.0159 ms
	Casanova 2	Yes	0.0098 ms
	C#	-	0.0147 ms
Unity3D	Casanova 2	No	0.0257 ms
	Casanova 2	Yes	0.0085 ms
	C#	-	0.1642 ms

In Chapter 4 we discussed the preconditions for a condition to become interesting. This is important, since the data structure supporting our optimization adds runtime overhead, and thus should only be used when the overhead is lower than the gained performance. Therefore, all attributes involved in the condition should exhibit some sort of temporal locality in order to benefit from our optimization.

Specifically, if the behavior of inappropriately optimized entities is not coherent with respect to the flow of time (in our case every entity would get activated too soon), the cost of entering and exiting the data structure implementing the *fast-wake-up* mechanism becomes too high, to the point that the performance gained is less than the costs, and therefore overall performance. To support this observation, in the following we discuss this with concrete numbers.

We made a series of variations to our initial scenario, to test the performance of our optimized code with different activation times. The activation times that we are going to test are: between 5 and 30 seconds, between 10 and 30 seconds, and between 15 and 30 seconds. We chose 5 seconds as the lowest activation time, since below this amount the performance of our optimized code is always worse than the non-optimized code. For this scenario, when activation time is below 5 seconds the condition stops being interesting. Moreover, for every scenario we tested a different number of initial entities (besides the 1000 entities added every 5 seconds). This increase in number of entities should add some dynamism to the application, as more entities will interact with the *fast-wake-up* data structure. Thus, by means of the Visual Studio profiler we can measure the costs of interactions between the different game entities and the fast-wake-up data structure. The initial number of entities in question are: 500, 1000, 3000, and 5000 entities. For this scenario, we used the MonoGame framework with .Net 4.5.

The results of this scenario are summarized in Figures 7.4, 7.5, 7.6, and 7.7. As we can see the optimized version performs better when the activation time of each entity is greater than 5 seconds. However, performance decreases when the scene becomes too crowded.

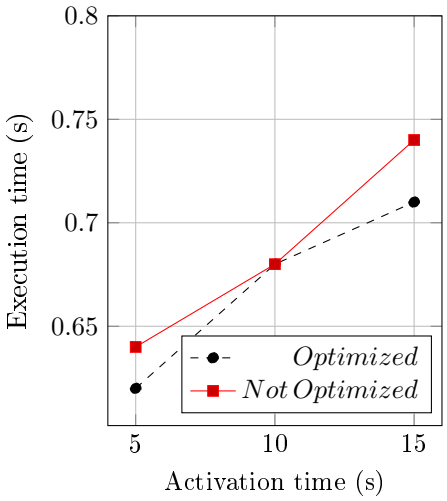


Figure 7.4: Comparing performance of optimized vs. non optimized Casanova 2 version, with 500 entities

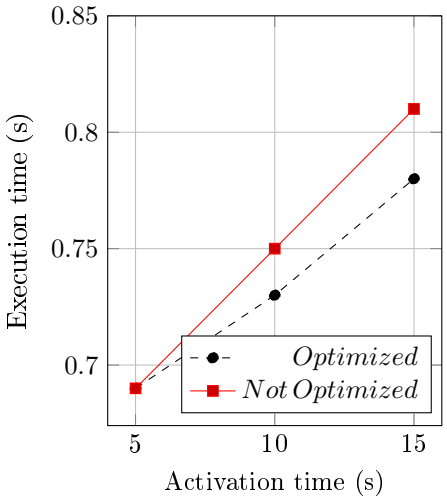


Figure 7.5: Comparing performance of optimized vs. non optimized Casanova 2 version, with 1000 entities

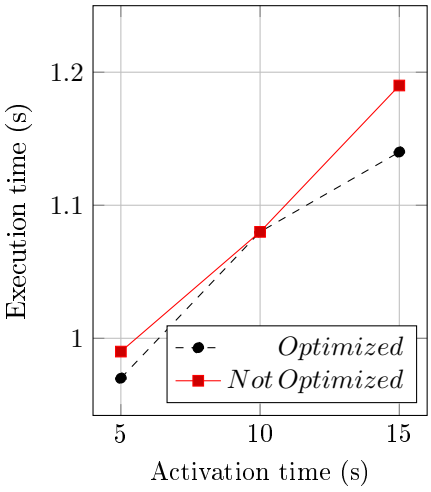


Figure 7.6: Comparing performance of optimized vs. non optimized Casanova 2 version, with 3000 entities

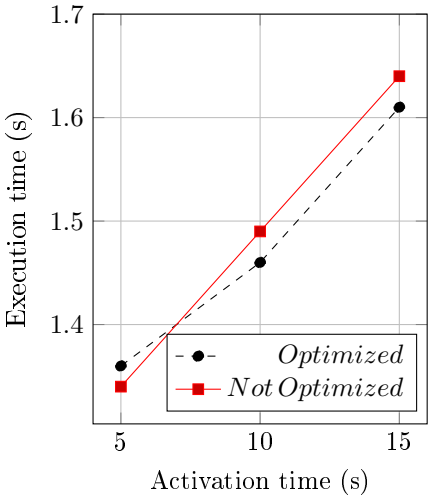


Figure 7.7: Comparing performance of optimized vs. non optimized Casanova 2 version, with 5000 entities

When the number of initial entities is greater than 5000 or the minimum activation time is too low (less than 5 seconds) the number of interactions with our *fast-wake-up* data structure becomes too high. After profiling the code with the above setup, we observed that activating the suspended entities costs less than 0.1% of the total amount of execution time of the application. Instead, handling the activated entities requires an increasing amount of resources.

In our implementation, all activated entities are stored in a dictionary, to speed up the check of whether an entity is activated or not. However adding, removing, and iterating the active entities come at a cost. The costs associated to these operations are: adding costs $O(1)$, but $O(N)$ in

case of a collision or in case we exceed the underlying arrays capacity; removing costs $O(1)$, but $O(N)$ in case of a collision; and iterating all entities costs $O(N)$. Thus, the more entities enter or exit the dictionary the higher will be the chance of a collision, thus the higher will be the costs. In our concrete implementation a method called `UpdateSuspendedRules` takes care of handling the activated entities, and to deactivate them whenever needed. In Table 7.3 we show the impact of the `UpdateSuspendedRules` method with respect to the initial amount of entities and their deactivation time. In this comparison we also added 1 second of minimum activation time to show the impact of interacting often with the dictionary.

Table 7.3: CPU activity of the `UpdateSuspendedRules` method in Casanova 2.

Minimum deactivation time (seconds)	Percentage of CPU activity in `UpdateSuspendedRules`	
	Initial amount 500	Initial amount 5000
1	1.76%	1.97%
5	1.35%	1.67%
10	1.27%	1.43%
15	0.85%	1.32%

As we can see the total percentage of execution time decreases as the activation time increases and the total amount of initial entities decreases. Indeed, as the amount of entities interacting with the dictionary decreases (either because the activation time of the entities is longer, or the number of entities interacting with our dictionary is less) the performance gets higher.

We also provide a detailed evaluation of this scenario in Table 7.4. In this table we find the exact measures of this evaluation (grouped by initial amount of entities and their minimum deactivation time). In each cell, on the lower left we find the total time that the optimized code took to complete the simulation (every simulation takes 2000 frames to finish, with a dt of 0.016 seconds), whereas on the upper right we find the total time that it took the non-optimized code to complete the same simulation.

Table 7.4: Detailed performance evaluation of the same running example run by Casanova 2 with and without optimization. For all the averages the amount of squared mean distance is less that 1.0E-3.

Minimum deactivation time (seconds)	Optimization enabled	Total average execution time (ms)			
		Initial amount 500	Initial amount 1000	Initial amount 3000	Initial amount 5000
5	yes	0.629	0.693	0.973	1.365
5	no	0.641	0.695	0.995	1.347
10	yes	0.689	0.738	1.080	1.465
10	no	0.689	0.759	1.087	1.49
15	yes	0.718	0.780	1.148	1.618
15	no	0.740	0.819	1.198	1.644

As we can see the performance of our optimization, when the deactivation time is below 5 seconds, is almost the same as the non-optimized version. This is due to the high number of interactions between the active entities and the fast-wake-up data structure. However, above 5 seconds the performance of the optimized solution is greater than the standard Casanova 2 solution. This is due to the fact

that entities stay deactivated for a longer period of time, and so their rules are not uselessly polled at every frame.

Our solution still exhibits good performance when the total number of entities interacting with the fast-wake-up data structure is greater than 5000. Moreover, we observe that, above this number, the performance of our solution decreases, due to accumulated costs of entering and exiting the fast-wake-up data structure. However, in a game it is difficult to find 5000 entities exhibiting temporal locality behavior. Most of game entities in a game are dynamic, and our optimization is meant for those that exhibit continuous and stable behavior. Thus, choosing the rules that exhibit some temporal locality, is a delicate task that should be derived from in-game observations. In the future we could delegate this task to a tool that analyses the game entities' behavior.

We also experimented for different settings the optimized implementation of the running example to test possible limitations of the data structure supporting our optimization. More precisely, we tried to fix the initial size of our dictionary to 10000 to reduce the number of possible collisions. Moreover, we also changed the fast-wake-up collection from dictionary to a sorted dictionary so as to have a constant addition and deletion time of $log(N)$. However, in both scenarios we did not observe significant changes in the results.

7.2.3 Readability

Building a video game requires encoding its design into a concrete program by means of some programming language. Every program allows many different encodings which all correctly represent the original design. These encodings vary in terms of variables names, structure of algorithms, structure of classes, etc. These encodings, while equivalent in the sense of all implementing the same design, are not all equal. When we consider readilibity in this evaluation, we first focus on how they range along a spectrum of complexity: some are simpler, while others are more complex. Developers always try to achieve those encodings that are optimal in terms of simplicity in order to keep the encoding simple and therefore cheap to maintain and extend. However, measuring whether an encoding is optimal is not possible in a fully mechanical, objective way. For example, Kolmogorov complexity is the common approach to discuss program complexity, but there exists no way to determine Kolmogorov complexity for practical programs [65].

Therefore, we are forced to resort to a series of heuristics to define some measure that we believe correlates with simplicity.

Ideally, a language should allow the definition of compact programs with the lowest amount of visible information that is not related to the problem, and with the highest amount of high-level information that is affine to the problem.

In this evaluation we used (*i*) the number of lines of a program as a metric to measure its length, but also (*ii*) the number of syntagms (i.e. the number of distinct keywords and operators) necessary to write the program [39].

i We counted the number of lines of each source implementing our running examples, thereby assessing the size of the implementation, with the assumption that a bigger sample corresponds to more complexity.

ii We counted the syntagms of each source implementing our running examples, with the assumption that a higher count corresponds to more knowledge required from the developer. Indeed, mental lookups of the sort "what does this operator do?" add overhead to our mind, and this makes the program harder to understand.

When combining these two metrics we can effectively measure the complexity of a program: a program is readable when we have both few lines of code and a low number of syntagms.

Table 7.5: Syntax comparison between the program written in Casanova 2 and the others written with the other tested languages.

Language	Syntagms	Lines of code	Total words
Casanova 2	47	31	104
C#	61	69	269
JavaScript	52	41	257
Lua	47	45	249
Python	50	34	214

It must be noted that some languages, which look compact at a first sight, come with a series of constructs that are difficult to manipulate and hard to understand. Indeed, when pushed to its limits, succinctness can damage readability of a programming language by ending up with the so-called "line noise" [48]. So, even if a program turns out to be short, this is not a guarantee that it is also readable.

In the following we show the results of our scenarios. We will assess the readability of the two scenarios introduced previously (Section 7.2.1). In the first one we show the readability of Casanova 2 and its constructs against other representative languages used in game development, which we used idiomatically. This evaluation shows how domain specific syntax and semantics can help keep code compact. The second evaluation presents the readability gained by Casanova 2 programs, against an idiomatic implementation in C#, by adopting the compiler optimization introduced in Chapter 5.

Scenario 1

For this evaluation we have taken the running example used in the first scenario of the performance evaluation section. The purpose of this scenario is to test how discrete and continuous dynamics in Casanova 2 can be expressed naturally within a block of code without any sort of adjustment, nor the need for special constructs. This is the reason why in Casanova 2 we find constructs for the manipulation of the flow of time in the program, which are natively and fully integrated in the language itself. This results in coherent syntax and semantics that can naturally express continuous and discrete dynamics. In the following we show the results of this choice and its impact in terms of readability and maintainability of Casanova 2 programs.

Results The results of this test are summarized in Table 7.5. As we can see, Casanova 2 resulted in significantly less lines of code and syntagms, especially with respect to C# (the only other language with comparable high performance).

These positive results for the Casanova 2 language are only possible because every its construct is designed to capture typical aspects of game development. For example, a rule (which represents a behavior of which the execution is repeated every time we reach the end of its block) can be suspended at any moment during a game. Now consider the following code in Casanova 2, which represents a suspendable rule containing two nested `for` loops:

```
rule ... =
  for x in [1..10] do
    wait 10<s>
    let z = random(0, 10)
    for y in [1..z] do
      ...
```

In the code above we have a `for` loop that at every iteration waits 10 seconds before recurring with the inner for loop.

When comparing the implementation above to others made with languages that do not natively support suspension of code, the resulting source will include low-level considerations, such as state machines, that will negatively affect the complexity and thus the readability of the code.

Indeed, if we would express the above code in C# then we would need a state machine capable of expressing the `for` loops, and the `wait` behavior. Moreover, since in the code above the variable `z`, which is a random number between 0 and 10, is used after its declaration in the inner for loop statement, the state machine would need a mechanism to store and propagate the intermediate generated variables, and make them available to the various blocks, or cases, of the state machine. As one can imagine this solution is not trivial to implement, and the reason is derived from the fact that C# was not designed to tackle this kind of issue as part of its idioms.

The other tested languages, such as Lua, come close to the code above, as Table 7.5 shows. However, these languages trade their generality for a loss of performance, since they translate the code above into a series of dynamic bindings represented with data structures each requiring a runtime representation (each binding binds the current block of instructions with its continuation).

Scenario 2

For this evaluation we have measured the lines of code and syntagms of the second scenario discussed in the performance evaluation.

More specifically, in this section we compare the code written in Casanova 2 against the output of the compiler, which is written in C#, and an equivalent version written in C# that does not include the optimization described in Chapter 5. For this scenario, it is important to focus on how much code a developer would write by hand in order to achieve comparable performance as in the optimized version, when such optimization is not supported natively by the language.

Results Table 7.6 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 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. Note that the intermediate code that does not include the optimization discussed in Chapter 5 is longer than the idiomatic C# implementation. This is due the presence of the state machines that the compiler generates to represent the rules bodies. Indeed, each rule is translated into a switch, for which the different cases capture the various blocks of a rule. Moreover, every construct in Casanova 2 that manipulates the execution flow of a rule is mapped to a series of constructs, such as `return` or `goto`, in the intermediate code.

Table 7.6: Code lines comparison for a single player game

Original language	Generated language	Optimized code	Lines
Casanova 2	-	-	45
Casanova 2	C#	No	139
Casanova 2	C#	Yes	327
C#	-	-	88

As we can see from the results in the above table, the number of lines of code belonging to the Casanova 2 version that includes the optimization discussed in Chapter 5 is considerably longer than the other implementations. This is due to the fact that the optimization itself comes with a series of additional data structures, of which the manipulation is not trivial. Indeed, at every game iteration

the world entity (after traversing all the entities) has to go through all the suspended rules that are active in that frame, and to run them all. When, after the execution of an active rule, the rule is `done` with its execution (i.e., it needs to get deactivated) the world entity will also be tasked to remove it from the active rules list.

Moreover, when an attribute is involved in an *interesting condition*, even more additional code is necessary. This code has to deal with the possible activation of the rules that contain interesting conditions that depend on the attribute in question. This gives an idea of how much more code should be directly written in C# to support such an optimization. No additional code needs to be included, since the compiler will add it automatically, as discussed in Chapter 5.

7.3 Summary

In this chapter we evaluated Casanova 2 with respect to the run-time performance and readability of its programs. For each of these evaluations we compared Casanova 2 against other tools for game development. Each of these tools is selected based on: (*i*) its relevance for the evaluation, and (*ii*) its representativeness in the community of video games development. The collected evidence shows that Casanova 2 is better than other representative languages used in game development in both readability and runtime performance.

We also discussed an analytical evaluation of Casanova 2, where we showed that Casanova 2 implements all the requirements, introduced in Section 2.3, that define the qualities that a language for game development should have. Moreover, as Casanova 2 is meant to work with any tool for game development, we presented a series of case studies where we showed Casanova 2 working with other third-party tools.

Casanova 2 works properly with other frameworks, is compact and maintainable, and has a compiler that produces fast runtime game code. In conclusion, Casanova 2 is a suitable language for game development that satisfies all requirements introduced in Section 2.4.

Chapter 8

Conclusion

This chapter provides a conclusive answer to the problem statement and research questions introduced in Section 1.4. Section 8.1 discusses the three research questions: what are the requirements for an ideal game development tool, to what extent a language can capture such requirements, and how such language performs in the reality of game development; Section 8.2 answers the problem statement, which discusses to what extent a tool that is built specifically for the domain of games can improve the process of making video games; Section 8.3 discusses future work; Section 8.4 adds the final remarks for this thesis.

8.1 Answer to research questions

The three research questions stated in Section 1.4 are now answered, in Sections 8.1.1, 8.1.2, and 8.1.3 respectively.

8.1.1 Game development tools requirements

The first research question reads:

> **Research question 1:** What are the requirements that an ideal tool for game development needs to meet?

The answer to the first research question is derived from Chapter 2. Specifically, in Section 2.3 we introduced a series of advantages and disadvantages which an ideal tool for game development needs to meet. Such advantages and disadvantages are derived from an analysis of the tools used in game development and from their evolution. In conclusion, the answer to the first research question is that an ideal tool for game development should:

- come with features to speed up the development process and to contain the complexity of game code,
- present its features in a way tuned to the domain of games, to make game code readable and therefore more maintainable,
- come with a series of already built-in strategies for increasing game code runtime performance, without the direct intervention of the developer,

- be able to interoperate with already existing tools and libraries available on the market, since some of these tools might come with closed solutions to some specific problems, and
- be able to build generic game genres, without any preference to a specific one.

At the same time such an ideal tool should avoid disadvantages that are inherent to several other game development tools. These disadvantages, which could be seen as antagonistic to the advantages above, are identified as: verbosity of programs, lack of portability, steepness of learning curve, lack of customization, low-performance, and gluing frameworks and libraries to compensate for a lack of fundamental design concepts, or constructs, in the adopted tool.

8.1.2 Implementing game development tools requirements

The second research question reads:

> **Research question 2:** To what extent can a programming language for game development be built, which meets the identified requirements?

The answer to the second research question is derived from Chapters 2, 3, 4, 5, and 6. In Section 2 we identified DSL's as a solution to the limitation imposed by GPL's used in game development tools, in order to tackle all those issues that are not (properly) tackled natively. In Chapter 3 we presented a concrete DSL called Casanova 2, which is aimed at reducing the complexity of making games. Casanova 2 comes with its own syntax and semantics that are designed around the common aspects of the video games domain. In Chapter 4 we presented a compiler that, together with ensuring structural correctness of Casanova 2 games, also translates Casanova 2 games code into executable programs with fast runtime performance. In Chapter 5 we further explored the opportunities offered by the underlying domain of video games to improve the performance of Casanova 2 games. In Chapter 6 we showed what a complete game looks like in Casanova 2. In particular, we showed that when embracing the requirements discussed in Section 8.1.1 at language level, the resulting games code is simple, compact, and readable.

8.1.3 Evaluation of the DSL

The third research question reads:

> **Research question 3:** How does such a programming language perform in terms of expressiveness, speed of execution, and maintainability, when compared to commonly-used tools for game development?

The answer to the third research question is given in Chapter 7. In this chapter we discussed the features and attributes of the Casanova 2 language by means of two different evaluations: a qualitative and a quantitative one, respectively.

In the qualitative evaluation we discussed the requirements specified in Section 8.1.1. For each of these requirements, we discussed how Casanova 2 accomplishes them by means of practical examples. In the quantitative evaluation we discussed the performance and readability attributes of Casanova 2 by a quantitative analysis, and we compared them with those produced by other representative tools used in game development. Casanova 2 was shown to outperform all compared tools and languages with respect to runtime performance, and compactness of its program's code.

8.2 Answer to problem statement

The problem statement reads:

> **Problem statement:** To what extent can a tool be built, which makes the complexities of general game development manageable for small and medium-sized teams of developers?

Our main goal in this thesis is to reduce the complexity of game code. We introduced *domain specific abstractions for game development*, and built *programming tools that implement those abstractions*. These abstractions assist developers in reaching their goals by substantially reducing development efforts, with a special benefit for smaller development teams that work on serious games. We wrap these abstractions in the concrete shape of a programming language.

A programming language suited for game development should always keep in mind the requirements discussed in Section 8.1.1. In this thesis we propose a programming language, called Casanova 2, which is aimed at achieving such requirements. Casanova 2 is a language specifically designed for building computer games, and it offers a solution to the high development costs and complexity of games.

Being tailored to the specific domain of games makes Casanova 2 capable of expressing properties that are common to the design of games, such as time flow, suspensions, games entities, etc. As an immediate result the resulting Casanova 2 code is similar to how we think of a game from a mathematical perspective (see Section 2.1). Moreover, Casanova 2 guarantees non-functional requirements (performance being the central one) as it comes with a series of built-in optimizations that do not require the developers' direct intervention.

Casanova 2 does not trade its advantages for expressive power. The language is not bound to specific genres, and is shown to be effective in the hands of a broad spectrum of developers.

We have shown these properties by means of a mixture of extensive benchmarks which compare Casanova 2 sources with equivalent sources written in typical programming languages, but also by means of actual game development activities performed by different developers, from junior to senior.

We consider Casanova 2 particularly suitable for smaller development teams, which are willing to trade some of flexibility and execution speed of general programming languages, such as C#, for speed and ease of development and maintainability of code.

Therefore, in answer to the problem statement, we state that Casanova 2 is a suitable language for game development that offers a significant step forward for developers in achieving a more efficient process of translating a game design into an actual, working game.

8.3 Future work

Casanova 2 comes with a series of features that have proven to be convenient for video game developers. However, a modern language for game development, such as Casanova 2, should also be flexible (which entails being useful in a broad range of contexts) and extensible (adaptable beyond the limitations of its original design). In order to be flexible and extensible, it is important that the compiler architecture is constructed in a way that is open to changes and extensions. This is not entirely the case for the current version of the Casanova 2 compiler.

Therefore, we are focusing our research efforts on the definition of a meta-compiler that supports the automatic creation of a compiler for any of a general class of programming languages. Preliminary results show that it is possible to implement an equivalent version of the Casanova 2 compiler in the meta-compiler, with less effort and lines of code when compared to the traditional hard-coded implementation.

This meta-compiler, once finished will allow us to experiment with adding new features to Casanova 2, without the need to make time-consuming changes to the existing compiler. Such features encompass networking capabilities, a dedicated debugger, and compilation towards different target languages.

Of these new features, networking capabilities are, in our view, the most important, and should be added to Casanova 2 soon. We already did some preliminary experiments in this regard, of which the results are reported in Appendix C.

8.4 The future of game development

Where once games were meant almost exclusively for entertainment, nowadays the applicability of games has expanded beyond that, and increasingly professionals who are not game developers themselves see a need or use for the development of a game for a particular purpose within their own knowledge domain. As such professionals usually do not have the budget to let their envisioned games be developed by dedicated game developers, they are in need of a game development approach that provides them with the ability to develop all kinds of games with compact, readable, and maintainable code that translates into fast executables. For this purpose, general programming languages are unsuitable as they are hard to learn and use, while most game development tools are limited to particular genres and produce relatively slow executables.

This is where domain-specific languages come in. A good DSL for game development provides a programming paradigm that is tuned to the development of games, allowing the developer to focus on the high-level game concepts (regardless of the game genre), safe in the knowledge that the DSL itself will take care of the concepts common to most games, and will produce fast executables. Casanova 2 is such a DSL.

Considering that games are ubiquitous nowadays, and are increasingly seen as playing a role in training, education, research, and social interaction, there is a clear need for Casanova 2 and its ilk. The existence of such a language may help in lifting the application of games in a variety of domains beyond the level of mere aspiration.

Appendices

Appendix A

Casanova 2 questionnaire

In this appendix chapter we discuss the experience gathered from the first workshop on Casanova 2. The workshop was held during the GameOn conference in Amsterdam on December 2015. A group of about 12 developers attended the workshop, which took about 3 hours. During the workshop the participants were invited to build themselves some samples in Casanova 2. All the materials used for workshop were provided through an on-line repository on Github (https://github.com/vs-team/casanova-mk2/wiki/Workshop).

A.1 Questions

At the end of workshop the participants attend a short survey. The goal of this survey was to understand the background of the participants, to understand whether the participants appreciated Casanova 2, and what are advantages and disadvantages of Casanova 2 observed by the participants.

In the following we provide the questions of this survey, and for each question we provide a motivation on why we chose it:

Q1 : **What is the best feature of Casanova?**

This open question is meant to investigate the observed and understood the advantages of Casanova 2 by the participant. It is also meant to investigate whether the observed and understood advantages coincide (or at least partially overlap) with the features that characterize Casanova 2.

Q2 : **What is missing?**

This open question is meant to investigate the observed missing features of the Casanova 2. Answers deriving from this question might help us with choosing future features to implement.

Q3 : **Would you consider using Casanova in your daily work?**

This closed question is meant to understand whether the observed, and understood, features of Casanova 2 are convincing enough to make new developers choosing for it.

Q3.1 : **If yes, for what kind of games?**

This closed question is meant to investigate possible and future applications of Casanova 2.

Q4 : **What computer languages do you use professionally?**

This open question is meant to investigate the professional developing background of the participant.

Q5 : **Which systems or tools have you used to make games?**

This open question is meant to investigate the professional gaming background of the participant.

Q6 : **What does the following code do? (answer: the color toggles between red and green ever second)**

This closed question is meant to understand whether the participant understood the basic mechanics of the Casanova 2 (including its syntax).

Q7 : **What does the following code do? (when the cube is selected it get scaled up until a countdown is over. Then the cube is destroyed)**

This closed question is more difficult than the previous one, and is meant to strengthen the answer of the previous question.

A.2 Grouped answers

In the following the grouped answers given by the participants are provided.

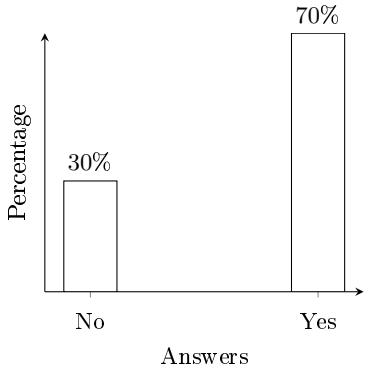
Q1 : **What is the best feature of Casanova?**

- Apparent simplification of threaded code execution
- Effective coding
- The coroutine like way of coding
- Interruptible statements
- Wait + condition seems elegant
- Abstraction of time behavior
- Casanova constructs more closely match a designer's intention
- A good environment for game development

Q2 : **What is missing?**

- Debugger
- Too little info to answer
- Too early to sya
- More precise parse errors notification
- Automatic generation of proxies for my custom entities
- A back end that compiles into C#/JavaScript/Python
- Do not know

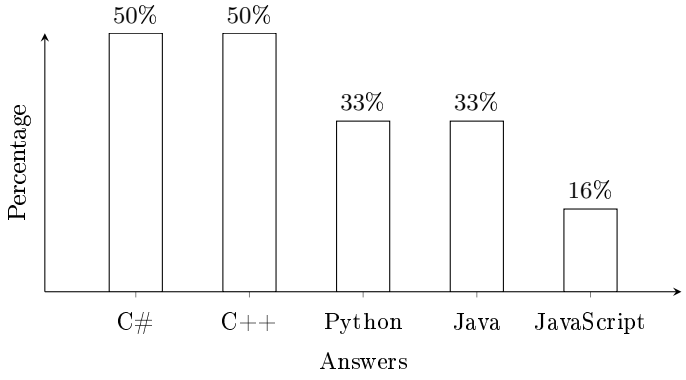
Q3 : **Would you consider using Casanova in your daily work?**



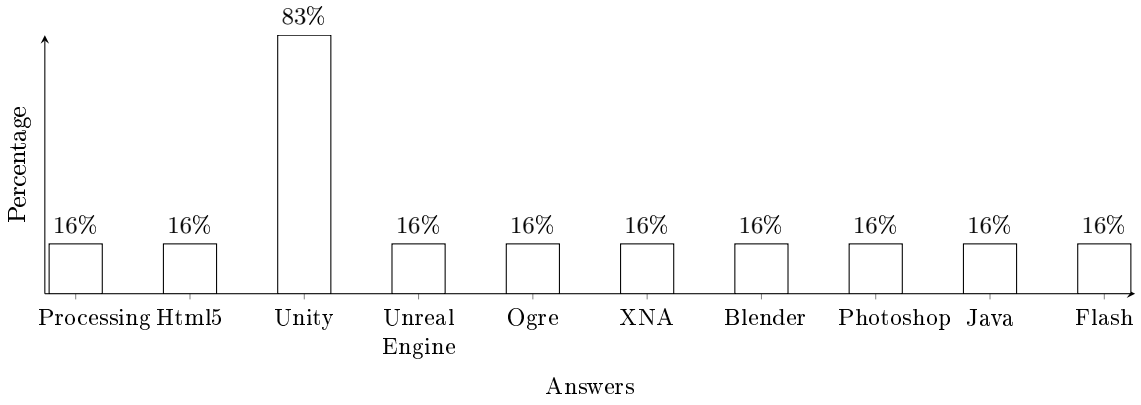
Q3.1 : **If yes, for what kind of games?**

- Language education games
- Educational games
- Education on game programming
- Puzzle learning for learning math

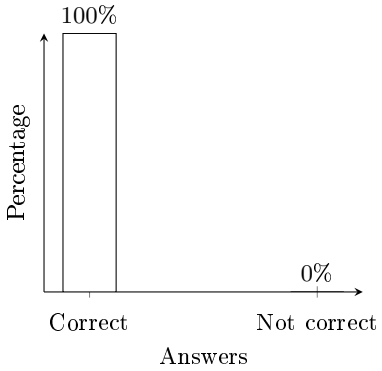
Q4 : **What computer languages do you use professionally?**



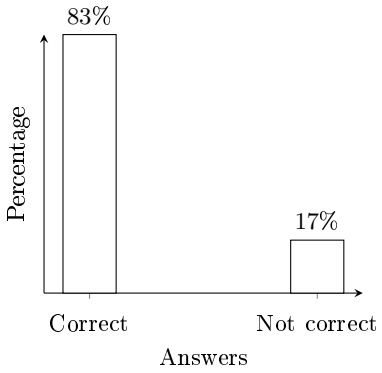
Q5 : **Which systems or tools have you used to make games?**



Q6 : **What does the following code do? (answer: the color toggles between red and green ever second)**



Q7 : **What does the following code do? (when the cube is selected it get scaled up until a countdown is over. Then the cube is destroyed)**



Results As a result, it turns out that Casanova 2 is appreciated by the experts who attended the workshop, although most of the participants would like to spend more time experimenting with the language. Moreover, all participants understood the advantages derived from the use of Casanova 2 such as compactness, readability, etc.

A.3 Discussion

Casanova 2 is appreciated by expert game developers (see questionnaire), who not only understood the advantages deriving from using Casanova 2, but also would use Casanova 2 in their everyday and professional lives. This questionnaire was not interesting to us only from the point of view of the appreciation level of Casanova 2; it also showed us that some import aspects are missing and felt by the expert developer. We are already planning to implement some of these aspects, such as a debugging facility, in on of the next development stages of Casanova 2.

Appendix B

Casanova 2 games

In this appendix we show how Casanova 2 works in real life. More specifically, we will show, and discuss, how Casanova 2 behaves when used by new developers who are not confident with it. We will do so by asking developers with no knowledge of Casanova 2 to build video games (our running examples). Moreover, by means of these games we wish to show how Casanova 2 provides a framework that is suitable for building video games not limited to specific genres.

In the following, we discuss these games and for each of them we will discuss its design, the technological choices made by the developers, and eventually our final observations.

B.1 Groups and games description

Every group is composed of third year bachelor students in computer science and who chose Casanova 2 as subject to work with

B.1.1 Group 1

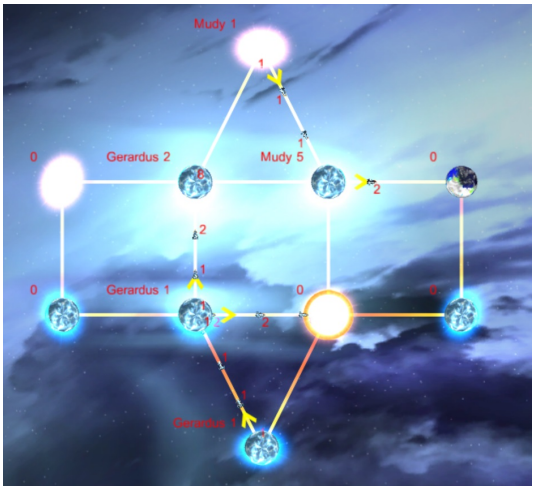
The first group is made of 4 students from the Rotterdam University of Applied Sciences. Before starting their work, the students received a short training on Casanova 2. After the tutorial, the students worked on a total of 2 games. The genres of games the students worked on are a strategy game, and a first person shooter game.

Game 1 - a strategy game

The first game the students made is a strategy games, inspired by the open source and online game *Galaxy Wars* (http://galaxywarsthegame.com/). In Figure B.1 you find a screenshot of the game made by the students.

Technological choices The students used Casanova 2 for developing the logic of the game, whereas Unity3D for: including the game contents, and rendering.

Our observations The game took the students a few weeks to be completed. When compared to the original game (the original Galaxy Wars logic was written in F#), the Casanova 2 code made by the students is more compact in terms of lines number.



(a) A strategy game made by a group of students

Figure B.1: A strategy game made by a group of students

Results The main mechanics we find in the original Galaxy Wars, expect for the networking (which is not supported in Casanova 2 yet), can be found also in the version made by the students. In conclusion, we can say that it is possible for novice programmers to make strategy games in Casanova 2.

Game 2 - a shooter game

The second game the students made, right after they previous one, is a first person shooter. The game belongs to the survival genre, where a group of players try to survive in a city overrun by enemy zombies by escaping from it on a virtual car.

Technological choices As for the previous game, the students chose Casanova 2 for developing the logic of the game, and Unity3D for: including the game contents, and renderings. Moreover, the game was developed so to be played inside a virtual reality lab (supplied with 360 degree projection), in order to let all the players play together in the same room. To simulate the weapons used in game, wired hand gestures controllers were used; while for driving the car the students used a USB racing wheel.

Our observations As observed before, when mastered, Casanova 2 speeds up the development process of a game. It took a short time (about 7 weeks) to implement the main functionalities of the game. Thus, leaving the students plenty of time for designing and implementing new additional features to include in the game. The main difficulties, we observed, encountered by the students are connecting and testing the external controllers and connecting the game to the virtual reality lab.

Moreover, it worth to acknowledge the fact that for this specific game, little technical support was provided. The students, all by themselves: designed and implemented the game, found suitable external controllers and connected them to Casanova 2, connected the game to the augmented reality lab, etc.

Results The game was completed in about 2 month and a half, and tested in the virtual reality laboratory together with all the external devices (steering wheel and gestures controllers). In conclusion, we can say that it is possible for novice programmers to make first person shooter games in Casanova 2.

B.1.2 Group 2

The second group is made of one student from Inholland University of Applied Sciences. As for Group 1, before starting with making games, the student received a short training on Casanova 2. After the tutorial, the students worked on a series of games, for a total of 6 games (5 of which are more simulations than fully fledged games), to run on a web browser.

Game 1 - tutorials

The games the student made vary from each other, since they are meant as examples for a tutorial in Casanova 2. The samples and games the student made are: a ship flying in the open space, a basket ball field simulation, a snowflake field, a flocking system, a series of controllable and moving patrols, and eventually an asteroids shooter game. In Figure B.2 you find the just introduced samples.

Each sample comes with a predefined learning goal that a new user, who is starting to study Casanova 2, would get:

- Moving ship, introduces the basics of Casanova 2;
- Basket ball field, teaches about interoperability with third-party tools;
- Snowflake field, introduces intermediate aspects of Casanova 2, such as how to manipulate entities in collections;
- Flocking, introduces advanced aspects of Casanova 2, such as how to build a physics system;
- Controllable patrols, introduces advanced integration aspects of Casanova 2, such as how to capture and propagate a click on a game element from Unity3D to Casanova 2;
- Asteroids shooter, sums up the acquired knowledge so far and puts it into the development of a fully fledged game that includes (together with the game dynamics) a menu system, audio effects, a scoring system, etc.

Technological choices As for the previous group, the student chose Casanova 2 for developing the logic of the samples, and Unity3D for: including the game contents, renderings, and running the simulations and games on a web browser.

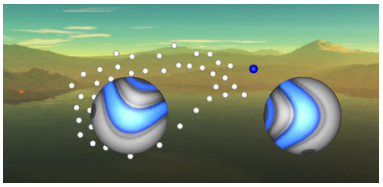
Our observations In about three months the student managed to implement, and comment (for the tutorial) all the 6 samples and games. As noticed with the previous group, the language was not difficult to use; most of the difficulties encountered by the student were related to other kinds of complexities, not related to the language itself, such as understanding the physics behind the flocking simulation.

Results All the samples were completed on time and are fully working and available via web browser on https://github.com/vs-team/casanova-mk2/wiki/Casanova Samples and Demos. In conclusion, we can say that it is possible for novice programmers to implement samples for tutorials on Casanova 2 and a space shooter.

(a) Moving ship

(b) Basket ball field

(c) Snowflake field



(d) Flocking

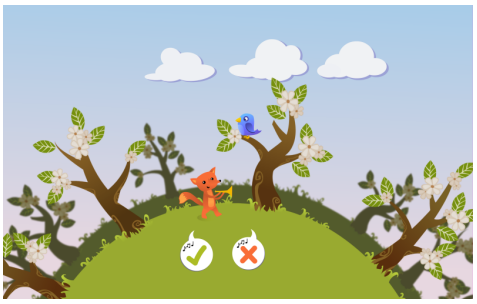
(e) Controllable patrols

(f) Asteroids shooter game

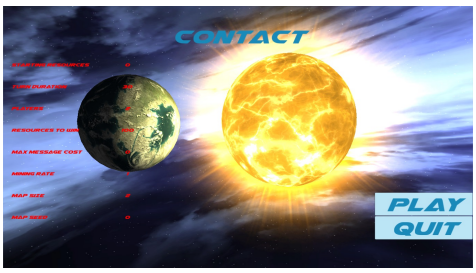
Figure B.2: A series of tutorials made by a student during his internship

B.2 Discussion

This collection of empirical evidences give a strong indication that Casanova 2 is suitable for making at least the games shown in this appendix chapter: first person shooter games, strategy games, space shooter games, and tutorial samples. Moreover, as observed previously, Casanova 2 code is compact and is understandable by novice developers to the point that novice developers managed in short time to implement different kinds of games.



(a) A game for detecting dyslexia in children



(b) A game for studying the evolution of a language

Figure B.3: Some Casanova 2 games

Casanova 2 has been used also for making a series of applications used as part of teaching and research projects. A notable application is a game for detecting dyslexia in children (see Figure B.3a). Another notable application is a game for studying the evolution of languages (see Figure B.3b). Both games have been used as a tool for research and features some articulated animations and state machines.

The collected results so far are preliminary (more running examples and users are needed to strengthen our observations), however these results are already interesting and promising.

Appendix C

Casanova 2 networking, a preliminary work

In this appendix chapter 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 game code compactness and maintainability. Typical networking implementations break encapsulation as 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.

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.

C.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 [47]. 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[104]) 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*[29]. Performance is both 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" [59]), 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 [89]. 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 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 the rely on reflection or events) or because they transmit too much data across the network.

C.1.1 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[18]. 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).

C.1.2 Related works

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 [46] 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 (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 [84] 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 [60] and Erlang [10]), which is periodically corrected by the server.

Unreal Engine/Unity Engine Unreal Engine [44] and Unity Engine [37] 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 (RPC's) of three kinds:

- The function is called on the server and executed on the client. This is used for game element 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 code.

C.2 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 fulfill these requirements.

C.2.1 The master/slave network architecture

We choose 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 architecture.
- 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.

Each client maintains a local copy of the `world` entity and has direct control over a single portion of it. Instances belonging to such 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 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.

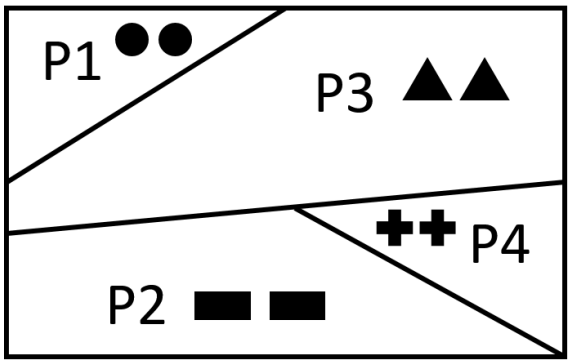
Casanova also provides primitives to send (reliably or unreliably) and receive data. A schematic representation of this architecture can be seen in Figure C.2.

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 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 [13].

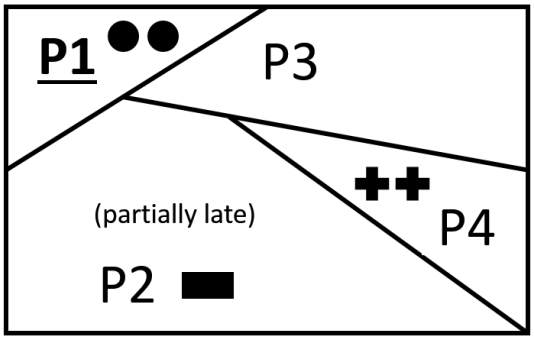
C.2.2 Case study

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 C.1:

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.



(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

Figure C.1: Representation of the game world in a networking scenario

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 modify 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.

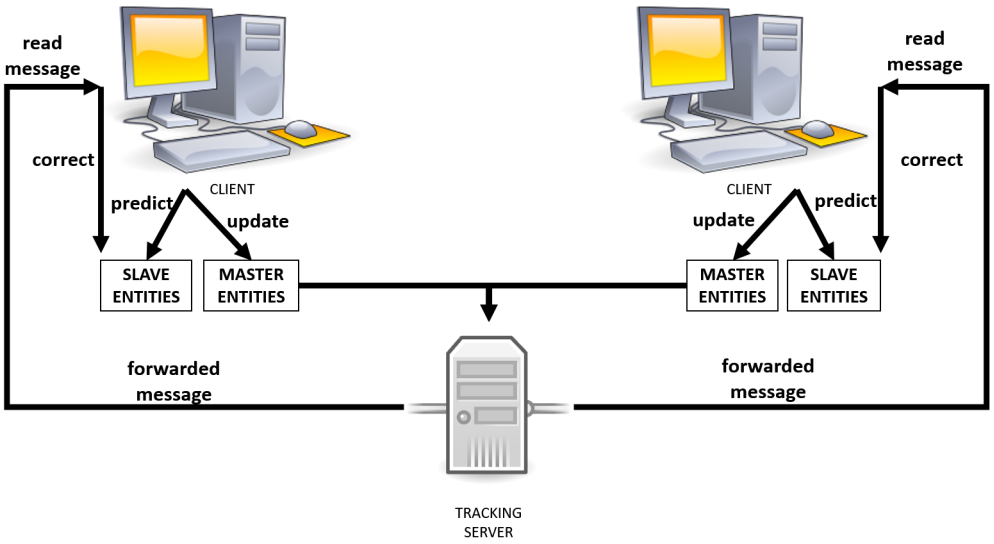


Figure C.2: master/slave architecture

C.2.3 Implementation

Each of the scenarios described above requires specific language extensions. This 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 [1]. We now give an implementation of the shooter game presented above and 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 = {
```

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

```
  Position : Vector2
  Velocity : Vector2
  ...
}
```

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. 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. 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
}
```

C.2.4 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.

- 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 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
}
```

- 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 it 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.

- 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. As explained in Section C.2, we only send the new projectiles and not the whole list.

```
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
}
```

C.2.5 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.

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

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

- 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
}
```

C.3 A preliminary evaluation

Compactness is an important aspect of a language that determines the maintainability of code written with it. Our proposal for a networking in Casanova 2 shows an interesting measure in terms of

Table C.1: Code lines comparison for a multiplayer game

Language	Lines
Casanova	126
C#	1257

compactness. When comparing the Casanova 2 game code of the networked game presented in Section C.2.2 (figure C.3 show this game in action and played by two clients) with an equivalent hand-made implementation written in C#, the difference is one order of magnitude (see Table C.1).

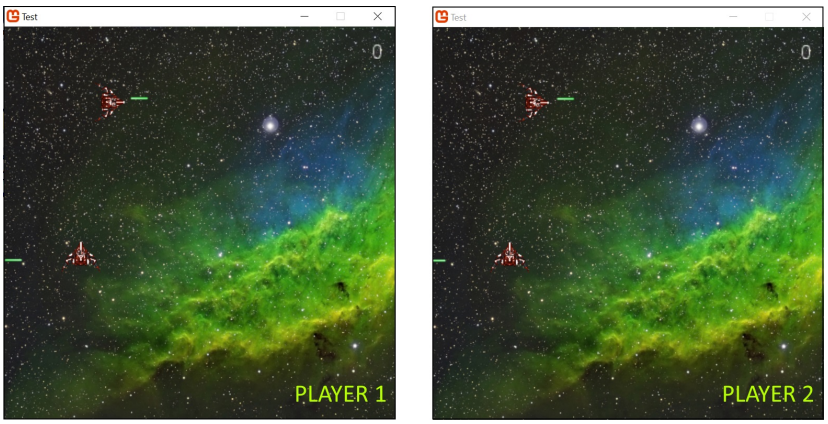


Figure C.3: A multiplayer Casanova 2 game made with the monogame framework

C.4 Discussion

Writing networking code by hand is a daunting and expensive task as seen in the tables C.1 and 7.6. To achieve the desired behavior, lots of code is necessary that will lead to a code that is less readable and maintainable. With our proposal the Casanova 2 is not affected by networking considerations, but rather the code remains compact, readable, and maintainable. These results are even more remarkable if we consider the fact that networking in games is known to be complex and typically require lots of verbose code. Our solution provides good primitives for networking, which preserve code encapsulation because they do not require polling 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.

Bibliography

[1] Performance evaluation code comparison. `https://casanova.codeplex.com/wikipage?title=Casanova%20Performance%20Comparison`.

[2] Essential facts about the computer and video game industry 2011, 2011.

[3] Galcon. https://www.galcon.com/, 2015.

[4] Mohamed Abbadi, Francesco Di Giacomo, Renzo Orsini, Aske Plaat, Pieter Spronck, and Giuseppe Maggiore. Resource entity action: A generalized design pattern for rts games. In *Computers and Games*, pages 244–256. Springer, 2014.

[5] Clark C Abt. *Serious games*. University Press of America, 1987.

[6] Damilare Darmie Akinlaja. *LÖVE2d for Lua Game Programming*. Packt Publishing Ltd, 2013.

[7] Tony Albrecht. Pitfalls of object oriented programming. *Proceedings of Game Connect: Asia Pacific (GCAP)*, 2009.

[8] Alan Amory, Kevin Naicker, Jacky Vincent, and Claudia Adams. The use of computer games as an educational tool: identification of appropriate game types and game elements. *British Journal of Educational Technology*, 30(4):311–321, 1999.

[9] Eike Falk Anderson, Steffen Engel, Peter Comninos, and Leigh McLoughlin. The case for research in game engine architecture. In *Proceedings of the 2008 Conference on Future Play: Research, Play, Share*, pages 228–231. ACM, 2008.

[10] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. Concurrent programming in erlang. 1993.

[11] Kendall E Atkinson. *An introduction to numerical analysis*. John Wiley & Sons, 2008.

[12] Alan D Baddeley, Neil Thomson, and Mary Buchanan. Word length and the structure of short-term memory. *Journal of verbal learning and verbal behavior*, 14(6):575–589, 1975.

[13] Bela Ban et al. Jgroups, a toolkit for reliable multicast communication. 2002.

[14] Francois Bancilhon. *Naive evaluation of recursively defined relations*. Springer, 1986.

[15] B. Bates. *Game Design*. Premier Press, 2004.

[16] Kent Beck. *Extreme programming explained: embrace change*. addison-wesley professional, 2000.

[17] Kate Berens and Geoff Howard. *The rough guide to videogames*. Rough Guides UK, 2008.

[18] S Bhatti, E Brady, K Hammond, and J McKinna. Domain specific languages (dsls) for network protocols. In *International Workshop on Next Generation Network Architecture (NGNA 2009)*, 2009.

[19] Lars Bishop, Dave Eberly, Turner Whitted, Mark Finch, and Michael Shantz. Designing a pc game engine. *IEEE Computer Graphics and Applications*, (1):46–53, 1998.

[20] Richard Blum. *Professional assembly language*. John Wiley & Sons, 2007.

[21] Ian Bogost. *Persuasive games: The expressive power of videogames*. Mit Press, 2007.

[22] Michael Buro. Real-time strategy games: A new ai research challenge. In *IJCAI*, pages 1534–1535, 2003.

[23] Michael Buro. Call for ai research in rts games. In *Proceedings of the AAAI-04 Workshop on Challenges in Game AI*, pages 139–142, 2004.

[24] John Charles Butcher. *The numerical analysis of ordinary differential equations: Runge-Kutta and general linear methods*. Wiley-Interscience, 1987.

[25] Paul A Carter. *PC Assembly Language*. Lulu. com, 2007.

[26] AT Chamillard. Introductory game creation: no programming required. In *ACM SIGCSE Bulletin*, volume 38, pages 515–519. ACM, 2006.

[27] HE Chehabi and Allen Guttmann. From iran to all of asia: The origin and diffusion of polo. *The International Journal of the History of Sport*, 19(2-3):384–400, 2002.

[28] Mark Claypool. The effect of latency on user performance in real-time strategy games. *Computer Networks*, 49(1):52–70, 2005.

[29] Mark Claypool and Kajal Claypool. Latency and player actions in online games. *Communications of the ACM*, 49(11):40–45, 2006.

[30] E. Collar Jr and R. Valerdi. Role of software readability on software development cost. 2006.

[31] J. Courtney. Using ant colonization optimization to control difficulty in video game ai. *Undergraduate Honors Theses*, 2010.

[32] Chris Crawford. *Chris Crawford on game design*. New Riders, 2003.

[33] Jason Darby. *Awesome Game Creation: No Programming Required*. Cengage Learning, 2008.

[34] CJ Data. *An introduction to database systems*. Addison-Wesley publ., 1975.

[35] Damien Djaouti, Julian Alvarez, Jean-Pierre Jessel, and Olivier Rampnoux. Origins of serious games. In *Serious games and edutainment applications*, pages 25–43. Springer, 2011.

[36] J. Eder, G. Kappel, and M. Schrefl. Coupling and cohesion in object-oriented systems. *Technical Report, University of Klagenfurt, Austria*, 1994.

[37] Unity Game Engine. Unity game engine-official site. *Online][Cited: October 9, 2008.] http://unity3d. com.*

[38] Carlo Fabricatore. Gameplay and game mechanics: a key to quality in videogames. 2007.

[39] Norman Fenton and James Bieman. *Software metrics: a rigorous and practical approach*. CRC Press, 2014.

[40] Ernest Ferguson, Brandon Rockhold, and Brandon Heck. Video game development using xna game studio and c#. net. *Journal of Computing Sciences in Colleges*, 23(4):186–188, 2008.

[41] Josef Fojdl and Rüdiger W Brause. The performance of approximating ordinary differential equations by neural nets. In *Tools with Artificial Intelligence, 2008. ICTAI'08. 20th IEEE International Conference on*, volume 2, pages 457–464. IEEE, 2008.

[42] Martin Fowler. *Domain-specific languages*. Pearson Education, 2010.

[43] André WB Furtado and André LM Santos. Using domain-specific modeling towards computer games development industrialization. In *The 6th OOPSLA Workshop on Domain-Specific Modeling (DSM06)*. Citeseer, 2006.

[44] Epic Games. Unreal engine 3. *URL: http://www. unrealtechnology. com/html/technology/ue30. shtml*, 2006.

[45] Arthur Gill et al. Introduction to the theory of finite-state machines. 1962.

[46] Frank Glinka, Alexander Ploß, Jens Müller-lden, and Sergei 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*, pages 81–86. ACM, 2007.

[47] Carl Granberg. *David Perry on game design: a brainstorming toolbox*. Cengage Learning, 2014.

[48] PJ Green. The apl alternative. In *GLIM 82: Proceedings of the International Conference on Generalised Linear Models*, pages 69–75. Springer, 1982.

[49] Bradley S Greenberg, John Sherry, Kenneth Lachlan, Kristen Lucas, and Amanda Holmstrom. Orientations to video games among gender and age groups. *Simulation & Gaming*, 41(2):238–259, 2010.

[50] Jason Gregory. *Game engine architecture*. CRC Press, 2009.

[51] Serge Guelton. *Building source-to-source compilers for heterogeneous targets*. PhD thesis, 2011.

[52] Tom Gutschmidt. *Game Programming with Python, Lua, and Ruby*. Premier Press, 2004.

[53] Tom Hastjarjanto, Johan Jeuring, and Sean Leather. A dsl for describing the artificial intelligence in real-time video games. In *Proceedings of the 3rd International Workshop on Games and Software Engineering: Engineering Computer Games to Enable Positive, Progressive Change*, pages 8–14. IEEE Press, 2013.

[54] Eva Hudlicka. Affective game engines: motivation and requirements. In *Proceedings of the 4th international conference on foundations of digital games*, pages 299–306. ACM, 2009.

[55] Johan Huizinga. *Homo Ludens Ils 86*. Routledge, 2014.

[56] ISO/IEC/IEEE. ISO/IEC/IEEE 24765 - Systems and software engineering - Vocabulary. Technical report, 2010.

[57] Meynaud Jean et al. Caillois (roger)-les jeux et les hommes (le masque et la vertige). *Revue économique*, 9(6):1001–1001, 1958.

[58] Simon L Peyton Jones. Compiling haskell by program transformation: A report from the trenches. In *European Symposium on Programming*, pages 18–44. Springer, 1996.

[59] Edward Kaiser and Wu-chang Feng. Playerrating: a reputation system for multiplayer online games. In *Proceedings of the 8th Annual Workshop on Network and Systems Support for Games*, page 8. IEEE Press, 2009.

[60] Harry Kalogirou. Multithreaded game scripting with stackless python. *Thoughts Serializer,(http://harkal. sylphis3d. com/2005/08/10/multithreaded-game-scripting-with-stackless-python/)*, 2005.

[61] Steven Kent. *The Ultimate History of Video Games: from Pong to Pokemon and beyond... the story behind the craze that touched our li ves and changed the world.* Three Rivers Press, 2010.

[62] Jan Willem Klop and RC De Vrijer. *Term rewriting systems.* Centrum voor Wiskunde en Informatica, 1990.

[63] David Kushner. *Masters of Doom: How two guys created an empire and transformed pop culture.* Random House Incorporated, 2004.

[64] R Kusterer. jmonkeyengine 3.0–develop professional 3d games for desktop, web, and mobile, all in the familiar java programming language, 2013.

[65] Ming Li and Paul Vitányi. *An introduction to Kolmogorov complexity and its applications.* Springer Science & Business Media, 2009.

[66] J Lloyd. The torque game engine. *Game Devel. Mag*, 11(8):8–9, 2004.

[67] Frank Luna. *Introduction to 3D game programming with DirectX 10.* Jones & Bartlett Publishers, 2008.

[68] Giuseppe Maggiore and Giulia Costantini. Friendly f# (fun with game programming), 2011.

[69] Giuseppe Maggiore, Alvise Spanò, Renzo Orsini, Michele Bugliesi, Mohamed Abbadi, and Enrico Steffinlongo. A formal specification for casanova, a language for computer games. In *Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems*, pages 287–292. ACM, 2012.

[70] Giuseppe Maggiore, Alvise Spanò, Renzo Orsini, Giulia Costantini, Michele Bugliesi, and Mohamed Abbadi. Designing casanova: a language for games. In *Advances in Computer Games*, pages 320–332. Springer, 2012.

[71] Christopher D Marlin. *Coroutines: A programming methodology, a language design and an implementation.* Number 95. Springer, 1980.

[72] Andy Marx. Interactive development: The new hell, 1994.

[73] Tom Mens, Michel Wermelinger, Stéphane Ducasse, Serge Demeyer, Robert Hirschfeld, and Mehdi Jazayeri. Challenges in software evolution. In *Principles of Software Evolution, Eighth International Workshop on*, pages 13–22. IEEE, 2005.

[74] Marjan Mernik, Jan Heering, and Anthony M Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.

[75] David R Michael and Sandra L Chen. *Serious games: Games that educate, train, and inform.* Muska & Lipman/Premier-Trade, 2005.

[76] Ian Millington and John Funge. *Artificial intelligence for games.* CRC Press, 2009.

[77] Jayadev Misra and William R Cook. Computation orchestration. *Software & Systems Modeling*, 6(1):83–110, 2007.

[78] Michael Morrison. *Beginning game programming.* Pearson Higher Education, 2004.

[79] Alan Mycroft. Programming language design and analysis motivated by hardware evolution. In *Static Analysis*, pages 18–33. Springer, 2007.

[80] Charles Petzold. *Microsoft XNA Framework Edition: Programming for Windows Phone 7.* Microsoft Press, 2010.

[81] William H Press. *Numerical recipes 3rd edition: The art of scientific computing.* Cambridge university press, 2007.

[82] Christopher Reich. Simulation of imprecise ordinary differential equations using evolutionary algorithms. In *Proceedings of the 2000 ACM symposium on Applied computing-Volume 1*, pages 428–432. ACM, 2000.

[83] Jeffrey Richter. *CLR via c#.* Pearson Education, 2012.

[84] George Russell, Alastair F Donaldson, and Paul 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*, pages 85–90. ACM, 2008.

[85] M Sebbane. Early bronze and middle bronze i board games in canaan and the origin of the egyptian senet game. *Eretz Israel*, 21:233–8, 1990.

[86] Manu Sharma, Michael P Holmes, Juan Carlos Santamaría, Arya Irani, Charles Lee Isbell Jr, and Ashwin Ram. Transfer learning in real-time strategy games using hybrid cbr/rl. In *IJCAI*, volume 7, pages 1041–1046, 2007.

[87] Dave Shreiner, Bill The Khronos OpenGL ARB Working Group, et al. *OpenGL programming guide: the official guide to learning OpenGL, versions 3.0 and 3.1.* Pearson Education, 2009.

[88] Miguel Sicart. Defining game mechanics. *Game Studies*, 8(2):1–14, 2008.

[89] Jouni Smed, Timo Kaukoranta, and Harri Hakonen. Aspects of networking in multiplayer computer games. *The Electronic Library*, 20(2):87–97, 2002.

[90] Shamus P Smith and David Trenholme. Rapid prototyping a virtual fire drill environment using computer game technology. *Fire safety journal*, 44(4):559–569, 2009.

[91] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *ACM Sigplan Notices*, 1986.

[92] Catherine Soubeyrand. The royal game of ur. *The Game Cabinet.< http://www. gamecabinet. com/history/Ur. html*, 2010.

[93] A. J. Stapleton. Serious games: Serious opportunities. In *Australian Game Developers Conference, Academic Summit, Melbourne*, 2004.

[94] Jonathan Steuer. Defining virtual reality: Dimensions determining telepresence. *Journal of communication*, 42(4):73–93, 1992.

[95] Steve Streeting and B JOHNSTONE. Ogre–open source 3d graphics engine. *URL-http://www. ogre3d. org/Accessed*, 26:09–12, 2010.

[96] Lists Strings. Backus-naur form. *Formal Languages syntax and semantics Backus-Naur Form 2 Strings, Lists, and Tuples composite data types*, 2010.

[97] Tarja Susi, Mikael Johannesson, and Per Backlund. Serious games: An overview. 2007.

[98] Albert Sweigart. *Making Games with Python & Pygame*. CreateSpace, 2012.

[99] Hirotaka Takeuchi and Ikujiro Nonaka. The new new product development game. *Harvard business review*, 64(1):137–146, 1986.

[100] Merel Tim. Global games investment review 2014. `http://www.digi-capital.com/reports/`, 2014.

[101] David Ungar, Randall B Smith, Craig Chambers, and Urs Hölzle. Object, message, and performance: how they coexist in self. *Computer*, 25(10):53–64, 1992.

[102] Peter Van Roy et al. Programming paradigms for dummies: What every programmer should know. *New computational paradigms for computer music*, 104, 2009.

[103] Mark JP Wolf. Genre and the video game. *The medium of the video game*, pages 113–134, 2002.

[104] Mark JP Wolf. *The video game explosion: a history from PONG to Playstation and beyond*. ABC-CLIO, 2008.

[105] Gang Zhou. *Partial evaluation for optimized compilation of actor-oriented models*. ProQuest, 2008.