Università
Ca'Foscari
Venezia

Corso di Laurea magistrale
in Informatica - Computer Science

Tesi di Laurea

—

Ca' Foscari
Dorsoduro 3246
30123 Venezia

Resources, Entities, Actions. A generalized design pattern for RTS games and its language extension in Casanova

Relatore
Ch. Prof. Renzo Orsini

Laureando
Mohamed Abbadi
Matricola 823145

Anno Accademico
2012 / 2013

To my beloved mother, and family.

Acknowledgments

To all the people that have been close to me and have helped me allong this journey.

To prof. Bugliesi for shaping me

To prof. Orsini for supporting me in this last stage of my study

To prof. Pelillo for changing the way I think about study

To Giuseppe for being a brother and a diligent teacher

Thank you very much

Abstract

In Real-Time Strategy (RTS) games, players develop an army in real time, then attempt to take out one or more opponents. Despite the existence of basic similarities among the many different RTS games, engines of these games are often built ad hoc, and code re-use among different titles is minimal. We created a design pattern called ”Resource Entity Action” (REA) that abstracts the basic interactions that entities have with each other in most RTS games. This thesis discusses REA and language abstraction and implementation using the Casanova game programming language. Our analysis shows that not only the pattern forms a solid basis for a playable RTS game, but also that it achieves considerable gains in terms of lines of code and runtime efficiency. We conclude that the REA pattern is suitable approach to the implementation of many RTS games.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

Video games have seen a remarkable growth adoption in the last decade [23], with game sales comparable to movie and music sales in 2010 [4]. This can be the main reason that justifies all the benefit and researches held in this field and the reason why many developers are involved in game development. Although games are generally associated to multimedia contexts and entertainment purposes, at the same time video games are used (through their simulations) even for serious purposes. The usage of video games is commonly not only in context of simple multimedia entertainment, but also in educational environments, training, etc. This because essentially a game is: ”an activity among two or more independent decision makers that are seeking solutions to achieve specific goals inside a bounded environment”. In essence we can summarize a simulation as a human being existence, although with differences such as simplifications or not realistic elements. This comes from the fact that the human is set to make decisions in a limited context to reach a specific targets . Decisions may be of different types ”when to buy a specific object”, ”when and where to go”. The context is limited, because the actions taken are produced in a limited resource, in a world where its variables are unknown or not completely clear. The objectives vary from managing resources to survive as long as possible, defeat an opponent, maximize a specific set of resources in a limited time, etc.

There are different kinds of games, but our research focuses on video games and particularly on strategy video games generally called real-time strategy games (RTS). Video games are a recent digital adaption of the commonly known genre of *board games*. Board games are frequently played in the real world and involve movable objects or predefined, in the real world game, following a set of predefined rules.

There are many different types and styles of board games that represent different life styles. For example: the chess game, which represents a specific battle field, while tic-tac-toe doesn't represent a specific theme. Rules may vary from very simple rules, such as those ones of checkers game, to very complicated and detailed, such as in Dungeons and Dragons. What makes Dungeons and Dragons a successful role playing board game is its game board, which makes it necessary for helping its players to visualize better the game scenario.

The definition of permitted actions and of the objectives to achieve, regardless of whether the game is serious or not, determines the involvement degree level of the game. For instance, surviving in real life is funny provided that it is challenging, fast and the application of its actions have big effects in the world. The same sets, but with slower actions, limited effects and played in real time may not result interesting from the entertainment point of view, but may result interesting from the educational point of view. Surviving in a game that requires efforts and competitiveness may results funny and challenging for the player.

The usage of video games for serious/educational purpose has been well known for time. Documentations report its usage since the 1900s [35]. Video games share a lot of common aspects with board games, except for the big difference, its digital platform. This implies that the game state management, the applying of its rules, the action controls are all managed by software components. Video games are often in real time, because the system provides actions with immediate consequences on both, game logic state and the visual one. Moreover many games represent complex scenarios with articulated logic that are difficult to transpose to the board games. This is possible thanks to the underlying software architecture which manipulates, manages, and maintains the game rules and their applications transparently for the players.

So the player can start playing a game without caring about its rule sets, but starts to approach and learn them during the game, throughout tutorials, experimentation, and trial-and-error. The real-time aspects, such as visual quality of simulations, big sets of actions and rules make games excellent candidates for representing simulations, with different degree of realism.

Video games are blends of the following different philosophies/aspects.

- Commercial games: it's the most important and comes for first. Commonly referred as AAA *games*, typically they are the most expensive, in terms money. It might cost millions of dollars for its development, and engages many years of developing and involves tens of professional developers.
- Indie games: this type of games are developed by small groups with limited resources; in these cases such games are referred as *IndiGame*.
- Serious games: it's a game of another genre, closer to the indie rather than the AAA, also known as *serious games*.Serious games are kind of games in which the player main goal is not entertainment but educational, awareness and instruction the player.

1.1 The challenge of game development

The cost budget associated to a video game creation are generally high, due to the resources involved, this is evident in AAA games, but it is as true (in terms of proportions concern) also for indie and serious games. The adoption of modern computers in the

80s, improved and renewed the movement of educative games introducing the educative video gaming. This genre is labeled as *edutainment* [16], deriving from the combination of the two terms *education* and *entertainment*. Examples of this genre are Oregon Trail, Math Blaster, and Number Munchers [13].

Serious games find a lot of difficulties in affirming, sometimes even during the development phase, because they are not economically profitable. The technological difficulties associated with its video game development don't attract investments because of its low economic return income. This has led, in years, to find different application environments for serious games. One of the first investors in this game genre is the United States Army, with major titles such as America's Army and Full Spectrum Warrior (these games are used by many users also for entertainment). It is interesting to note that military officers use intensively these games for strategic simulations. A recent game of this genre used for this specific purpose is *Kriegsspiel*.

Similar problems are taken also for the indie-games [25]. The indie games are sold, generally sold for less than 10 dollars (or are based on advertisement), and are often played through mobile phones, tablets, and on browsers. These are fully fledged games, but they are made simpler than those of the big industries [41]. The effort on the development of these titles by the indie-games developers is bigger than those of big houses. This is due to the limited resources (both humans and economics) of the indie-games developers. This brought to an increase of interest, from the research fields, of patterns and designed techniques to cut down the costs, and allowing these developers to test and explore new genres without risking big funds. Moreover the innovation made by serious games may be of big impact for what concerns the educational methods, a noble goal that with time it would bring profits to the whole system. The low cost development strategy impacts, both on serious and indie games development and reduces the innovation distances between them. This bring us to observe the *significant* importance of indie and serious games, and the reason why our focus is centered on these two typologies rather that the AAA games. The goal of this thesis is to reduce the risks of video games making (focusing on strategy games), reducing the efforts not only of development but also those of maintenance. We believe that doing so, by introducing a design pattern and its implementation through a language extension, we reduce many development patterns and activities that are recurrent in games. Obviously we do not pretend to solve all the problems of video games making, but in particular, if the design, or the selected algorithms are not of sufficient quality, the game will fail even if the use technology simplifies parts of the development.

1.2 RTS games

Real-time strategy games (RTS) have been highly popular for decades. As outlined by the ESA, RTS's are registering good sales and with a large number of playing hours. Commercial RTS games are written by different kinds of developers: from *large studios*, to smaller independent developers of *indie games*. Indie developers are a growing phenomenon within the video game industry [27]. Indie developers are typically small teams and their games are known for innovation [30], creativity [31] and artistic experimentation [15]. RTS games are also built outside the entertainment industry: many serious games [14] and research games [21] are RTS's. Studios which make serious games create interactive simulations which have as main purpose the training and education of users.

1.3 Problem statement and research question

Building games in general and RTS's in particular, are expensive[22]. This puts pressure on all game developers, but especially on indie, serious game and research game developers because of their smaller resources. This lack of resources motivates research in the direction of cost-effective development methodologies for games. Such research may aim in reducing development efforts through the identification and automation/reuse of common patterns in games and their automation in order to promote reuse. Making a new game could then leverage part of the effort of past games, and not just the knowledge and experience which must still be applied for every title. Surprisingly, from a survey of game development research and literature we noticed a lack of abstract pattern study which characterizes these games, and particularly with RTS's. This motivates our research question: *what are the common patterns of RTS games, and how do we capture them?*

1.4 A pattern for making RTS games

RTS are a variation of strategy games where two or more players achieve specific (often conflicting) objectives by performing actions simultaneously in real time.The typical elements which arise from this genre are *units* (characters, armies), *buildings*, *resources* and *battle statistics*. Players command units to perform different types of actions. These actions can affect several entities in the game world.

Units and buildings are the entities that the players control to achieve their objectives. Units usually fight or harvest resources, while buildings may be used to create new units or research upgrades. Resources are gathered from the playing field and the fuel is the economy of the game entities. Battle statistics determine the attack and defense abilities of units in a fight. This taxonomy of the elements of a RTS game can be applied successfully to multiple games: Starcraft, C&C, and Age of Empires all feature units, buildings, resources, and battle statistics, among other elements.

In order to arrive at our design pattern we will now apply a simplification. Battle statistics can be interpreted as resources, so that ”the life of a unit is the cost for killing it, payable in attack power”. We can also merge units and buildings together into a new category called *entity.* This leads us to a simpler view of a RTS as a game that is based on Resources, Entities and Actions:

1. Resources: numerical values in the battle and economic system of the game. In this group we find the *attack*, *defense*, and *life* patterns of entities. Resources also cover building materials and cost of production, deployment of units, development of new weapons, etc. (Resources are scalars.)

2. Entities: container for resources. They have physical properties and, as for the game logic, the difference among them is only the interactions. These interactions take place with resource exchanges through the actions. (Entities are vectors.)

3. Actions: The resource flow among entities. Our model can be viewed as a directed weighted graph where the nodes are the entities, the weights are the amounts of exchanged resources, and the edges are the actions, that is, the elements which connect entities to one another. (Actions are transformation matrices.)

This leads us to a more precise research question: *how do we model Resources, Entities and Actions?*

1.5 Structure of this work

We'll view in Section 2 the general structure of a game. In section 3 we'll view the available systems and languages for making video games, in particular for RTS games, emphasizing their pro and cons for video game development. In section 4 we'll show how to capture the essential elements of an RTS. The design pattern studied, ensures that developers can reuse some significant parts of game logic across multiple games, but using it requires effort from the developer. This effort consists of writing boiler plate code that links the definition of the game world and the design pattern primitives. To go even further, eliminating the writing of boiler plate code, we extended the Casanova game development language to automate this pattern (Section 5, 6, and 7). Casanova is a newly designed computer language which integrates knowledge about many areas of game development with the aim of simplifying the process of engineering a game. The language extension is purely declarative. Its semantics resemble SQL, providing an intuitive adoption for most programmers. Extending the language, gives us control over the way the game world is explored and over RTS mechanics; this allows us to build additional mechanisms such as automated optimization, which increases performance at no additional cost. We show some implementation of some RTS games (section 8) and evaluated this extension in a full-fledged RTS (Section 9), and compared programming effort and run time efficiency.

This paper presents the following contributions. Our technique is shown to reduce the amount of boilerplate code that needs to be implemented, tested, and debugged, therefore yielding increased development productivity. In this paper we provide evidence that our approach is (*i*) syntactically simpler, (*ii*) semantically powerful, (*iii*) general purpose, (*iv*) high-level, and (*v*) high-performance.

Chapter 2
Requirements of a game

Making video games offers the same core challenges that appear in all games. A game is a real-time application which simulates a virtual environment, in which the user can interact with a set of predefined actions. This interactivity is due to a main structure, which is known as game loop. The game loop continuously call functions which run the game logic and then draw the world and if the loop is fast enough, then the sensation will be of a real-time interaction. Game computes a numerical integration of the game state. Such integration is made from the world's physics, the input of the user, the artificial intelligence (AI) and the behaviors of the game entities. AI and most of the game logic are made by state machines in order to implement loops, timers, etc. The game also draws the game world so that the user can see a visual representation of the entities of the last few hundredths of seconds. Among the big challenges that developers incur during the making of a game are:

- Should it run on different platforms
- Multiplayer inside a real-time game requires reliable synchronization through the network
- The visual simulation must be realistic, detailed, and believable
- The simulation of the world and its entities must be enough complex and believable
- Inside a game project there is a creativity portion made by the designers, whom many times don't know much about computer programming so the architecture

of the game should be flexible enough and easy to use in order to facilitate such people to test, build new game iterations of game-play.

This list of challenges is not complete, but these elements are very common and discussed in many seminal texts [26] [17]. As expected these elements are the result of our direct experience in this field. Moreover, the costs of making video games are increasing as well as the hardware complexity too. As a consequence of these new possibilities of making more realistic games (real-time multiplayer, more challenging AI, more interaction among game entities, very detailed visual elements and realistic physics) are available. Let's consider a small scenario, to show how the complexity of a game component increases as well as the other components increase their complexity? When the game is very simple (it simulates just few components) then building an AI capable to interact with the player comes to be very easy to build, but if the game starts to be complex with very articulated aspects which involves many components, then the AI must take into the account more elements than before and this makes making such AI times more difficult. Later we will discuss how these features are built in a modern video game.

2.1 Game Loop

The game loop (often referred as principle loop) is responsible for updating the internal structure that represents the world game, both for the internal rules of the world and externals input from the player. After updating the structure of the world, the game loop draws the world entities on the screen (every frame the elements on the screen are erased and rendered anew). The motivation of using this draw model is given by the fact that the game elements generally move around the world during the game session, in order to render anew object every frame instead of checking if the element has moved, save a lot of overhead from the point of view of the performances. A simple implementation of the game structure could be given (we use a pseudo ML code) by just an update and draw function. The update function accepts only the elapsed time, so we can compute on how much the system should update the objects.

```
type GameState =
{
        Update:  float -> unit
        Draw: unit -> unit
}
```

At this point we can give the definition of the game loop function which takes as input the game state and alternately calls; first the update and then the draw function. The update function takes as its argument the elapsed time given by the difference between the current and previous frame. This fraction is used to compute correctly the integration of the update function and all the entities with respect to the appropriate amount of time:

```
let gameLoop (game:GameState) =
        let rec AUX_loop (prev_time:GameTime) =
        let  new_time = getTime()
        let dt = new_time - prev_time
        do game.Update dt
        do loop new_time
        do AUX_loop (getTime())
```

Indeed, a game can be seen as a numerical integrator that integrates the following integral for all time T of the game: $\int_{t=0}^{t=T} \mathrm{dwt/dt}\,\mathrm{d}t$

Where wt denotes the state of the world at time t. Let us consider a world made of a single soldier who has velocity, position, and acceleration.

```
type World=
{
        SoldierVelocity : Vector2
        SoldierPosition : Vector2
        SoldierAcceleration: Vector2
}
```

Integrating the world, means that we compute (and then draw on the screen) the game state at time $t = 0, t = 1/n, t = 2/n, ..., t = n/n$ for a game that runs n frames at second. Of course instead of calculating the integrated game world at a certain time from every frame, we make an incremental computation so we have to perform only the missing part of the integration. This means that since the above integral is impossible to compute analytically, instead of computing it every time t starting from $w0$, we compute wt from $wt - dt$. The integration of this new concept, called Euler method, in the previous update definition would then be:

```
let update (world:World) (dt:float)=
{
        SoldierVelocity =world.SoldierVelocity + world.
           SoldierAcceleration * dt
        SoldierPosition =world.SoldierPosition+world.
           SoldierVelocity * dt
        SoldierAcceleration=world. SoldierAcceleration
}
```

Because the integration of this integration is important and the frame rate maybe slow in some machines, with low computational power, sometimes the integration accumulates errors. To avoid this other integration method, its more precise to use, for instance the Ralston's one.

```
let update (world:World) (dt:float) =
        let f(v_old,x_old) == (world. SoldierAcceleration, v
           )
        let k1 = f(world. SoldierVelocity, world.
           SoldierPosition)
        let k2= f(world. SoldierVelocity, world.
           SoldierPosition) + 0.5 * dt * k1)
        let k3= f(world. SoldierVelocity, world.
           SoldierPosition) + 0.75 * dt * k2)
        let v_new, p_new =
                (world. SoldierVelocity, world.
                   SoldierPosition) +
                (2.0/9.0)*dt*k1+(1.0/3.0)*dt*k2+ (4.0/9.0)dt
                   *k3
        {
        SoldierVelocity = v_new
        SoldierPosition = p_new
        SoldierAcceleration = world. SoldierAcceleration
}
```

The above shown update function is more precise than the Euler, when dealing with physical objects such as balls, collision detection, etc. and the approximation errors are under the perception threshold of the user. Unfortunately this method does more operations than the other one and it is more difficult to write and maintain so it is not always the best solution: in some cases the simpler version may be better. This is one of the many trade-off between numerical accuracy and run-time speed for which the game loop should take into the account during the update function. It can provide exception for games which require high simulation quality, for instance some combat simulations or real life simulation. The previous two samples show a simple scenario made of just one physic simulation. For this reason it is easy to understand how to apply the numerical integration. A simple AI may be integrated as well, even though it must be threaded as a hybrid system (with both the aspects continuous and discrete). For instance in the following world:

```
type World =
{
        Enemy = Entity
        Self = Entity
}
```

The enemy can choose to attack or hide depending on the relative strength of the player.

```
let update (world:World) (dt:float)=
        let self_new = update_entity world.Self
        let enemy_new = update_entity world.Enemy
        let enemy_new' =
                {
                enemy_new with
                Velocity =
                        If enemy_new.Health > self_new.
                           Health  &&
                                    distance(self_new, enemy_new) <
                                       10.0 then
                                          towards enemy_new self_new
                               else
                                          away enemy_new self_new
                      }
        { Self = self_new; Enemy = enemy_new' }
```

The enemy of the previous example is a simple reflex agent, which depending on the situation he decides to attack or not. This solution suffers from the fact that if the player continuously enters and exits from the enemy area, the enemy will change continuously its behavior. To avoid this problem we can add a timer so that the enemy will maintain its action until the timer is over. Of course we can make a more complex AI, but this is not important for our presentation. Instead it is important to underline the presence of discrete components in the game that makes the numerical integral more complex to calculate. The game loop is not straightforward as in the previous example. To not waste resources, it is not necessary to update continuously all the elements of the game objects. While physical objects must be updated at every frame of the game, while other components as the AI and the user input can be updated with low frequency. For example AI can be run 5 times per second, while the input can be run 20 times per second all this without losing interactivity; this technique reduces much the computational load. To express different update frequencies, we need to change the definition of the game loop presented above in the following new representation:

```
type GameState =
{
        UpdateInput : float -> unit
        UpdateAi : float -> unit
        UpdateLogic :float -> unit
        Draw : unit -> unit
}
```

The game loop will run now three different loops: one for the Ai, one for the Input and one for the Logic for the game logic and the drawing of the game. A multithread loop can run the distinct loops.

```
let game_loop (game:GameState)=
        let rec loop f t dt =
                While(gameTime() - t < dt) sleep(0);
                Do f dt
                Do loop f (t+dt)
        let logic_draw_loop()=
                Loop (fun t -> game. UpdateLogic dt; game.
                   Draw())
                getTime()
                (1.0/60.0)
        let ai_loop() = Loop game.UpdateAi (1.0/5/0)
        let input_loop() = loop game. UpdateInput (1.0/20.0)
        run_thread [logic_draw_loop;ai_loop;input_loop]
```

In the above code the function loop performs a generic function with a certain interval of time defined by *t* and *dt*. So we use this function to run the three update functions, update ai, input, and logic/draw.

2.2 State Machines

The second very common activity in games is the maintaining of the state machines. State machines are used in games which use timing, ai, input, etc. to the point that, a simple web search shows that there are hundreds of tutorials and manuals about this theme and many book and articles that talk about this topic. Let's consider a simple example where we want to wait a specific event before doing something else. In particular we want to wait that a particular key is pressed in order to fire with a specific weapon. We can code this function as an extension of the previous update function which does some polling:

```
while (is_key_up(key_one)) do sleep(0)
let time = getTime()
while (is_key_up(key_two)) do sleep(0)
let dt = getTime()-time
If dt < 0.3 then
        do fire_weapon()
```

Unfortunately we cannot run this piece of code inside the update function, because this will lock it. We can run it in a separate thread, but if all the functions do so, the number of running thread would be too much decreasing, so the overall performances. To solve this issue the comon solution is through state machines which explicitly track where we are in the code. We then need a discrete step function which updates, at each step of the game, the state of the state machine, managing its state and checking the various transition conditions. The above machine can be defined as an ML discriminate union.

```
type StateMachine =
        |Wait
        |WaitT of float
        |Done

        |Fail
```

And the update state machine function updates the state machine as follows:

```
let updateStateMachine (sm:StateMachine) (dt:float) =

        match sm with

        | Wait ->

                if (is_key_up(key_one)) then do sleep(0.0)

                else Wait

        | WaitT(t) ->

                if t <= 0.0 then Failure

                elif (is_key_up(key_two)) then Success

                else WaitT (t-dt)
```

The general pattern for state machine with states $s1, ..., sn$, transition matrix ti, j (that defines the condition for going from si to sj), we can define the state machine as:

```
type StateMachine =

        | S1 of s1

        | ...

        | Sn of sn
```

The transitions are defined as:

```
let transition (sm:StateMachine) (dt:float) =

        match sm with

        |S1(s1) ->

                if t11 then S1(i11)

                elif t12 then S2(i1n)
```

```
          ...
          else S1(s1)
    |...
    |Sn(sn) ->
          if tn1 then S1(in1)
          elif tn2 then S2(in2)
          ...
          elif tnn then Sn(inn)
          else Sn(sn)
```

2.3 Drawing

The game logic gives life to the interactive simulation, where the game world and its entities are updated according to some specific internal rules, AI, and physics while the user can change the state thorough his contribution. Of course though, the user need a way for recognize the state of the world for making a plan and deciding his next moves. Allowing the user to know the game state is done by visual elements on the screen. The visual elements are typically textures, 3d models, but some notable (though not very common) exceptions simply write a text description of the current state of the game world to the screen which the user then reads. Such games usually are referred as text adventures and are not our main focus. Drawing is done in a manner that it acts as a window over the game through which the user watches the game entities in real-time. We draw the game world after few logic updates, so that the visual differences will be very small between each frame. Usually logic entities are mapped one-to-many to the visible entities, which can be textures, 3d models, fonts, etc. It is important to notice that most of the richness and complexity of a modern game doesn't come from having many kinds of visual element on the screen; drawables elements are almost 7: text, textures, or 3d models. To distinguish such elements we use different effects each of them specialized to do a specific job.

2.4 Summary

In the previous sections we identified the base requirements of a game, which are:

- managing the game loop
- managing of the input
- state machine support
- drawing the scene

It is important to notice that commercial high quality games (often known as AAA games) need also other requirements. Large games have lots of assets produced by artists and designers, and so they need tools that support transferring those assets into the game engine. Designers also produce game scripts that customize some behaviors of the game entities, which require data-centric engine architecture so that scripts may customize large parts of the game without direct access (a complex task) and re-compilation (a surprisingly lengthy task) of the main sources. Sometimes the effort made by remaking all the visual aspects of the game starting from zero is excessive; and the reuse of components and libraries for different titles is not an easy job to do. In this thesis our will is not be on this kind of games but instead on the indie/serious-games which are smaller and more maintainable.

Chapter 3
Available game development systems and languages

In this chapter we'll be comparing the choices between programming languages and systems form making in video games. We start by giving a background of systems and languages and the comparison between them according to their advantages and disadvantages. We will be discussing the pragmatic reason of why most of developers choose systems instead of languages for making video games. Moreover, we will discuss in detail some notable game development systems, and choose three of them for their significance and current adoption. We then will be doing the same for the programming languages. From our survey and considering the points underlined in the last chapter we can conclude that there is the need of more exploration of specialized languages for developing video games, because these can give significant advantages that actually are not exploited by game developers.

3.1 System vs languages

A system form making video games is a collection of libraries and tools that are used for making video games. A game development system offer mainly predefined functions that the developer uses whenever he needs. For instance, using an editor, a developer can drag and drop functionalities over a model just using a mouse or a keyboard. Game development system often offers some customization due to the support of scripts written by the developers. Such scripts are small programs that modify limited aspects of the game, for instance the formula that computes the damage in a weapon, personalized triggers, timers, etc. The scripts do not modify the base services of the game. Some game developments systems are mainly based on libraries, which are then accessed through a programming language. Pre-build functionality are then used by invoking the library or by instantiating its data type. Such libraries are supported by external tools, for instance custom project settings for popular development environment. A game language is a programming language which is used for making video games. The syntactic constructs of a game development language are directly mapped to the abstract aspects of the game. For instance a game development language may have specialized syntax for drawing the game world, in this way it helps in reducing the amount of notions that the developer should know and reduces the amount of code, in this way it helps the developer to explore the functionality through a visual interface. Languages, on the other hand, should help the developer to focus on the problem instead of the implementation details, offering the maximum expressive power and reducing the amount of obvious and repetitive code. Languages and systems offer advantages over one another. Generally systems represent a secure investment where we leverage high-quality components built by others, while taking advantage of existing technical knowledge to the fullest. But systems often have limitation in power expressiveness. Such limitation can be of different shapes. It is possible that most of the games build with a specific system can be in some way very similar to each other. Languages, instead, represents a risky investment where we spend a lot of energy to express the different solutions for the problems. The knowledge acquired during the process of developing video games lets the developers to develop quickly, making fewer errors in game by game, and making code increasingly clearer. Languages for making video game offer a power expressive power, but on the other hand they are harder to learn. The systems are more secure to use, and for this reason most of the companies use them for making video games instead of the computer languages. The rise of many high-quality commercial game development systems in the last decades means that there are a lot of systems of undoubted quality. Languages for making video games, on the other hand, are not vey used if not in the academic environment. So the adoption of commercial languages for making video games is insignificant.

3.2 System for making games

It's out of our scope discussing all the available systems involved in games making. Available systems cover most of the types of games: many systems are specialized for first person shooter, role player games, flight simulators, and so on for the other kinds of games. This happens because the games represent interactive virtual realities. A virtual reality is a real time simulation with its rules, its logic, its goals, and its means of interaction. Such virtual reality can be close enough to a real life for some aspects. The number of ways that a virtual reality can be expressed is big: any aspect, from the physics, to rendering, to the introduction of new elements and schemes may be changed by our will. This implies that the range of possible games are very wide, with the following consequence: for many games, it is very difficult to find existing systems that completely satisfy our creativity requirements and sometimes a system that initially seems to satisfy our requirements may fail during the process of making the game due to the addition of new game elements. This comes from the fact that systems for making video games are very strict and support just specific scenarios. There are systems which support just first person shooter games, systems that simulates just fast cars races, systems specialized on rendering of characters in third person view as for the role playing games, systems specialized on rendering a very big number of elements as for the war strategic games, and so on. As we see now, we will list some game systems that are very common; we then choose three relevant systems for further study.

3.2.1 Relevant game systems

There are many systems for making games. The earliest examples of engines used for many games include 2d games were built in the 1980s. Among these, Adventure Construction Set [1], Garry Kitchens game Maker [5], Wargame Construction Set [12], Pinball Construction Set [6], Shoot Em-Up Construction Kit [8], Arcade game Construction kit [2], and ASCIIs RPG Maker engine that are still nowadays being released [7]. The 1990s, saw the introduction of the first graphics engine; BRender [3], Render-Morphics Reality lab [42]. The term game engine started to be used in the second half of the 1990s specifically in relation with FPS games. In particular, games like Quake and Doom were so popular, that the developers authorized the use of some parts of their code for making other games (levels, graphics, etc.). The engines listed above cover just a small part of the available game kind; there are other kind of games such as the adventure one, platform, fly simulations, etc. each of them associated to specific engine that satisfies their requirements. Games that are not supported by any engine require new engines that support them. Among the most important systems we also find low level frameworks such as OpenGL [20] and DirectX [28], which simply offer a set of libraries in addition to tools such as profiler or debugger, which can be easily used from existing programming languages and IDE.

3.2.2 Our choice of systems

We choose three representatives of the actual trend:

- Game Maker [19] is a system for making video games based on the simple philosophy of ”less is more”, in this way it allows beginners to make articulated games.

Game maker is a visual environment which limits the developer to build only 2D games based on camera and 2D sprites which represent objects, characters, etc.

- Unity [34], on the other hand, is a powerful and flexible game development system which also, gives the possibility of making 3D games. It offers many generic components which can be combined with the entities in order to customize them. Unity allows expressing many different scenarios.
- XNA [34], is a framework oriented for making indie games based on the .Net framework. The systems we consider are chosen for representing both game engines and libraries. We consider them relevant because they are used much by the communities. We exclude engines such as UDK and Quake, because they are used for making AAA games, instead our thesis focuses on serious and indie games.

3.2.3 Game maker

Game maker is an IDE which allows users to easily make computer games without requiring big experience, allowing at the same time expert developers to make complex games through its built-in scripting language. The main interface of game maker is a drag and drop. The available menu allows the creation of a hierarchy structure of sprites which represents the room where the game takes place and the entities will inhabit the game world. Available icons represent the common actions which are of interest in a game, as movements, basic drawing, e simple flow control which allow the creation of reactions as the detection of a collision. The drag and drop method, allows the definition of simple games, but it is a limited system when an advanced user tries to define more complex components such as an AI, path findings, etc. Game maker has a built-in language called Maker Language (GML). GML can access and modify most of the values of the game, we can write cycles, and in general it is a fully fledged

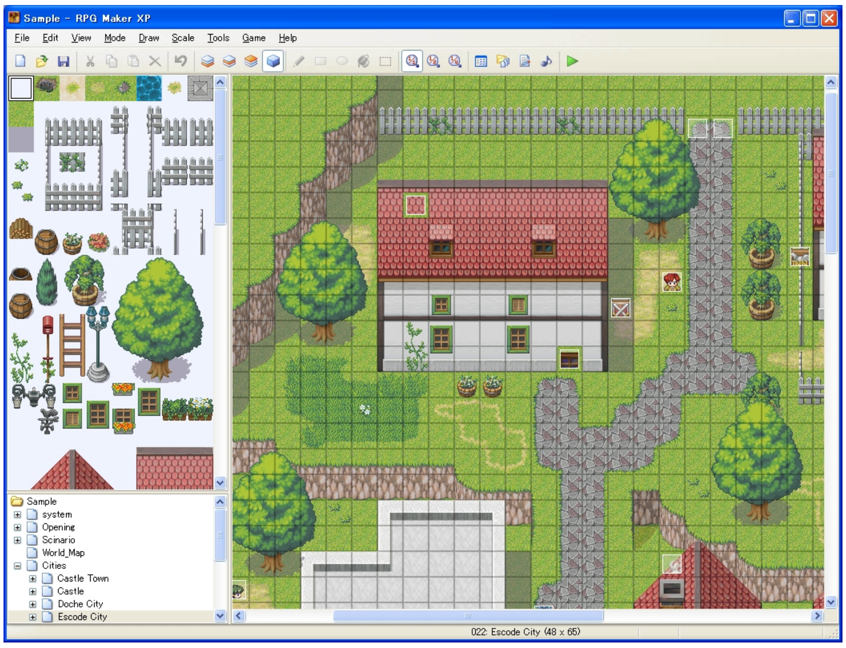

Fig. 3.1: *Game maker in action*

programming language. Game maker interpret through an interpreter the games. Game maker is mainly based on 2D graphics, with a variety of expected functionalities, more than simple drawing of sprites, such as alpha adjustment and blending settings. Game maker can be seen in figure 3.1.

3.2.4 Unity 3D

Unity 3D is an assets-centered game system. This means that Unity tries to keep the developer as much as possible in the visual editor, dragging new game entities in the hierarchy of the scene, and modifying the objects properties to customize their appearance or behavior. The editor also shows a live preview of the game. Unity 3D supports high quality 3D. The underlying architecture is based on components. Components are properties of the game entities which allow a certain entity to perform certain rules. Components uses physics to respond to an input command, and so on. When a developer selects an entity in the visual editor, he then attaches components to

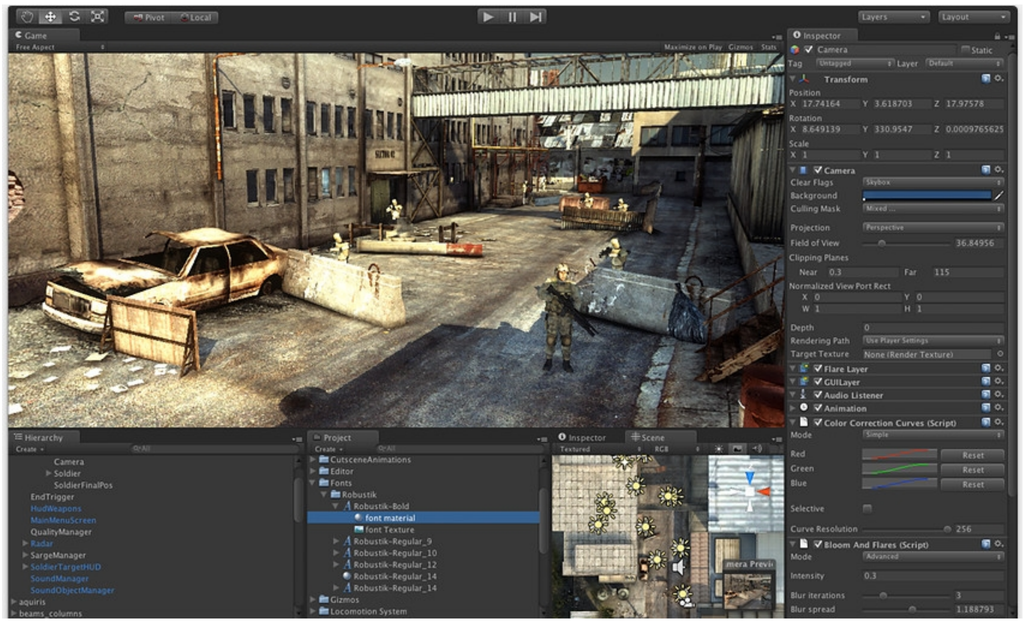

Fig. 3.2: *Unity 3D in action*

it. As for game maker, the visual editor is not enough in expressing complex behaviors. So Unity supports personalized script through Mono, an open source implementation of the framework .Net. The scripts may be written in Unity Script (a personalized language inspired by JavaScript), in C# or in Boo [24]. Unity 3D can been seen in figure 3.2.

3.2.5 XNA

Game maker and Unity mainly include a scripting system so that the game can be managed through scripts written by the developer. These scripts track just some part of the game state such as score, life, AIs, etc. and can be ran either as reply of an event or at each step of the game. This may result in contrast with the visual nature of the editors, which aim in allowing users making games without writing any line of code. This will lead us to say that the definition of a game is intrinsically linked to the coding; it is not important how the visual editor is powerful there is always the need to write some functionality by coding it. Visual editors can help users to reduce the amount of written code. Continuously new algorithm are invented and published. This means that, for a game system it is impossible to follow and express each of these algorithms. In short, game development requires coding. For this reason we take into account XNA which is a tool based on coding for making games based on XNA and that can be used by any .Net language: from c# to F#, to VB.net. It offers classes which cover most of the areas of game development. Such classes represent an efficient and easy to use and understand implementation of the components common in all the games. XNA stars starts with the game loop and game components classes to support the game. The input is supported by polling the mouse, keyboard, and game pad. To ease building collision detection and physics, a series of bounding volumes are supported such as bounding volumes, bounding boxes, etc. XNA also offers renderer facilities to let the developer to add quickly 2D sprites, and 3D models. The user moreover can customize the rendering by writing its own shades for specific visual effects. Audio is supported with music, positional 3D sound, and even pre-mixed audio effects. Networking unfortunately is not completely supported. XNA does not offer any visual framework or helper but instead requires coding the whole game, often with a lot amount of coding. XNA can be seen in figure 3.3.

3.3 Languages for making video game

There are not many special purpose languages for making video games and the possible causes are of different kind:

- Making a special purpose language is complex and requires specific knowledge of the environment (in our case games) and parser on which the language is applied.

Fig. 3.3: *XNA in action*

- Systems for developing video games compensates the lack of expressive power through by integrating scripting languages.
- Game sectors are slow in adopting new programming languages compared to the IT industry, this had led in the last decades to move first from the assembly language, then to C and finally to C++

This is due to the big amount of already done code built in C/C++ and that is still used in production. Outside the game environment, some researchers have experienced the construction of special purpose language for making video games, while the industries still focus on libraries coded in C or C++ and in the recent years in C#. Languages used for making games built by researchers are all specialized distinctly in the aspects of the game developments, as a result of this, there is a small set of languages specialized for game development. We will discuss four of these languages and in particular:

- Simula [32], a very old language built for simulations, from which many languages have taken inspiration.
- SGL [40], a recent developed language from the Cornell University which unifies

the game looping definition with a series of much optimized SQL queries on a global table which memorize all the game entities

- Inform7 [33], a language concentrated on just a small kind of games, the ones on textual adventures, but allows to write them in plain English, instead of a traditional programming based approach

- Casanova [11] is a programming language designed around games. Casanova is a language for developing videogames which allows building of games with three important advantages when compared to traditional approaches: simplicity, safety and performance. Casanova is a research project based on Giuseppe Maggiore's PhD thesis. There are various Papers that discusses this language and written by many researchers such as me, Francesco Di Giacomo, Enrico Steffinlongo, Michele Bugliesi, Pieter Spronck, Renzo Orsini, and Aske Plaat.

3.3.1 Simula 67

The first language we discuss is Simula, in particular version Simula 67 (exists 2 version of Simula: Simula1 and Simula67, the second one is more advanced than the first one). Simula is an old language and was the precursor of the Object Oriented languages. It was made in the 1967 and it is specialized in making simulations, as the name suggests. In particular, Simula is characterized by objects, inheritance, and cooperative (through coo routines) multi-threading. Simula is historically an important language, because its object system laid the groundwork of many languages as C++, Java, and C#. Moreover Simula uses very heavily coo routines which are widely used by many modern languages for instance in making interactively collections. Coo routines in particular are used much in the C# language in the shape of async. Also before C#, Mono and Unity have introduced a personalized version which uses coo routine through the keyword yield.

Now let us see an example of Simula67 in action. Mudy, Giulia, and Giuseppe are going shopping for clothes. They have to share the dressing room. Each of them is searching for about 15 minutes in the shop and then uses the dressing room exclusively for 5 minutes. A simulation of their dressing room is the following:

```
Simulation Begin
   Class FittingRoom; Begin
      Ref(head) door;
      Boolean inUse;
      Procedure request; Begin
   If inUse then Begin
         Wait(door)
         Door.First.Out
      End;
      Procedure leave; Begin
         inUse := False;
         Activate door.First;
      End;
      Door :-New Head;
End;



Procedure report(message); Text message; Begin
   OutFix (Tim2,2,0); OutText("␣:␣␣" &message);OutImage;
End;


Process Class Person (pname); Text pname; Begin
```

```
   While  True Do Begin
      Hold (Normal(15, 4, u));
      Report(pname & "␣␣is␣requesting␣the␣dressing␣room" );
      Fittingroom1.request;
      Report(pname & "has␣eneted␣the␣dressing␣room");
      Hold(normal (3,1,u);
      Fifittingroom1.leave;
      Retport(pname & "has␣left␣the␣dressing␣room");
   End;
End;
Integer u;
Ref(FittingRoom) FittingRoom1;
FittingRoom1 :- New FittingRoom;
Activate New Person("Mudy");
Activate New Person("Giulia");
Activate New Person("Giuseppe");
Hold(100);
End;
```

The keyword at the beginning of the listing enables the simulations. The dressing room uses a queue called DressingRoom which arbitrates the entrances of the room. If somebody asks to enter in the room and it's occupied, then he will wait (Wait(door)) until its being released. When the dressing room is emptied, the user in the head of the queue is activated (Active door.first) and removed from the queue. A person in modeled as a process and its activities are described by the keyword ”Hold” which suspends the process to simulate its activities in the shop. The program then simulates for 100 minutes the games by initializing the entities at the beginning. Until today no other languages has reached the expressiveness of Simula. The sample above for instance, would be harder to express for a general purpose programming language as C++, given the lack of first class support for cooperative multi-threading, queues, etc.

3.3.2 Inform

The Inform 7 language is the last incarnation of Inform; it is a programming language and design system for interactive fiction developed in the 1993 from Graham Nelson. Inform7 is a peculiar programming language; it addresses a niche-within-a-niche of the domain of computer programming. Inform exclusively allows the developing of interactive text adventure games, where the world of such game genre are represented by description texts and the users interact with the world by once again, text commands. This makes Inform a specific language for making specific kind of games. The advanced features of Inform allow the programmer to code the game in English, although with some grammatical limitations. The language allows the description of the game, a collection of objects with properties, together with a collection of scenes which interact with the user. The parsing facilities offered by Inform are very powerful and are used to parse the text input of the user which is expressed in natural language. Inform allows single hierarchy and polymorphism. The language is strongly typed in order to let the programmer to reuse the code, without adding complex concepts as abstraction, so the entities definitions in the game are simple. This means that some traditional programming activities are supported by the language, but all the constructs are declarative and highly readable.

```
Constant Story "HELLO␣WIKIPEDIA";
Constant Headline "^An␣Interactive␣Example^";
```

```
Include "Parser";
Include "VerbLib";

[ Initialise;
    location = Living_Room;
    "Hello␣World";
];

Object Living_Room "Living␣Room"
    with
        description "A␣comfortably␣furnished␣living␣room.",
        n_to Kitchen,
        s_to Frontdoor,
    has light;

Object -> Salesman "insurance␣salesman"
    with
        name 'insurance' 'salesman' 'man',
        description "An␣insurance␣salesman␣in␣a␣tacky␣
           polyester
␣␣␣␣␣␣␣␣␣␣␣␣␣␣suit.␣␣He␣seems␣eager␣to␣speak␣to␣you.",
        before [;
            Listen:
                move Insurance_Paperwork to player;
                "The␣salesman␣bores␣you␣with␣a␣discussion
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣of␣life␣insurance␣policies.␣␣From␣his
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣briefcase␣he␣pulls␣some␣paperwork␣which␣he
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣hands␣to␣you.";
        ],
    has animate;

Object -> -> Briefcase "briefcase"
    with
        name 'briefcase' 'case',
        description "A␣slightly␣worn,␣black␣briefcase.",
    has container;

Object -> -> -> Insurance_Paperwork "insurance␣paperwork"
    with
        name 'paperwork' 'papers' 'insurance' 'documents' '
           forms',
        description "Page␣after␣page␣of␣small␣legalese.";

Include "Grammar";
```

Inform does not have many constructs, which are necessary to build complex real time simulations: there is not any concept of game loop, any language for defining real time actions, or a way to customize the rendering. From the point of view of making video games it is woefully under-powered, but when it comes to interactive interactively fiction it is, a unique experience to use. Many games have been built with Inform. Curses, was the first game written with Inform, Galatea is considered to have the most complex interaction system, Mystery House Possessed, and are just few of the available games written with this language.

3.3.3 SGL

The last language that we will discuss is SGL. SGL, like the name suggest, is a language that derive from the well known SQL language for expressing declarative queries on databases. SGL is based on the idea of defining the world as a big empty table where an entity is a role and the columns represent all its attributes. The attributes that are not related to the entity are set to null. The game dynamics, as physics, damage, etc. are defined as queries which transform the entities during the simulation. For instance in SGL game virtual soldier's entities can be defined as follow:

```
Class Unit{
   State:
      Number unit_id;
      Number player;
      Number command;
      Number pos_x, pos_y;
      Number health;
      Set<number> squad;
   Effects:
      Number move_x : AVG;
      Number move_y : AVG;
      Number damage : SUM;
      Set<number> joined : UNION;
      Set<number> left : UNION;
   Update:
      Pos_x = pos_x + move_x;
      Pos_y = pos_y + move_y;
      Quad = squad UNION joined SETMINUS left;
      Health = health - damage;
}
```

The state encodes the entity description and all its attributes. The effects are a set of queries that are specified below. The update consists on how the state of each entity changes during the game simulation. The query which moves a unit towards enemy clusters may be defined with an SQL-style aggregation such as:

```
Let enemies = (all u in Units where u.player != me.player);
Let  centroid_x = AVG(enemies.x_pos);
Let centroid_y  =AVG(enemies.y_pos);
Let me.move_x = (me.pos_x - closest.pos_x)/norm;
Let me.move_y = (me.pos_y - closest.pos_y)/norm;
```

SGL games maintain most of the readability of SQL queries, that is, even for simple games not simple and pleasant to read as for the Inform game. But it is more pleasant to read when compared to others low level languages. The algorithmic optimizations are done by the system which relieves the programmer from the job of coding complex pieces of codes. SGL allows developers to make games without warring about performances but obtaining the runtime performances those results from the use of hand written, specialized, complex algorithms. From this point of view SGL is not completely a game programming language but it allows building efficiently the core module of a modern game.

3.3.4 Casanova

Computer games are seen as the next frontier of entertainment, with game sales being comparable to movie and music sales in 2010 [9]. This unprecedented market prospects and potential for computer game diffusion among end-users has created substantial interest in research on principled design techniques and on cost-effective development technologies for game architectures. But making games is an extremely complex business. Games are large pieces of software with many heterogeneous requirements, the two crucial being high quality and high performance [10]. High-quality games are measured by two factors: visual quality and simulation quality. For visual quality many researchers have pushed the boundaries of this field, and continue till today. Simulation quality instead is often lacking in modern games. The end result is that, while the visuals of a game are often stunningly realistic, the artificial intelligence of the game entities are not animated by high quality AI. Input controllers are used in a simplistic way and the logic of game levels is more often not completely linear. Along this direction, the necessity to reduce the costs as well as increase the quality of the simulations has led us to Casanova. Casanova is a language for developing video games which allows the building of games with three important advantages, compared to traditional approaches: simplicity, safety and performance.

- safe underlying model for updates based on double buffering: make less mistakes that depend on update order
- units of measure: no mix ups of a Vector2<m> with Vector2<m/s>
- integrated co routines for simpler state machines, AIs, and timers
- automated save game facilities: saving and loading requires just one line of code
- first class menu management

- nested drawing for easier positioning and expression of visual dependencies
- first class resources to implement economy or complex damage/defense systems
- first class physics integration

Chapter 4
The Casanova language

In this chapter we describe informally the design of Casanova. We start by talking of a recurrent and fundamental pattern in video that Casanova captures, which the authors called rule-script-draw (in short RSD). Then we show the goal of this language, and in the end we present informally the language.

4.1 The RSD pattern

The concept of RSD captures completely the definition of a game. Through our experience in developing video games, we realized that always the making of a video game touches some fundamental concepts. These can be modeled as 3 principal components. The game mainly is made by logic and drawing. The logic which is made of regular aspects of the game, such as physics equations, and others continuous parts of the numerical integration of the game, and irregular aspect, such as timers, AI, state machines and other discrete parts. And finally, many aspects of the logic are draw on the screen, in order to let the user to see them. So these three aspects of the RSD pattern are:

- *Rules*, the regular continuous aspect of the game
- *Scripts*, the discrete aspect of the game
- *Drawing*, the drawing of the game on the screen

Unfortunately, most of the games are developed without an explicit awareness of these components. This means that efforts to capture these so that building new games in them requires less time and expenditure of resources are not well focused if they do not consider such aspects. For instance, the XNA framework or DirectX UT offers (although in a limited way) rules and drawing, but any support for scripting is given. More advanced systems such as Unity 3D, offers more support of scripts through coroutines. Casanova captures the RSD pattern, through a full complete programming language with its compiler. Some attempt to capture recurrent patterns have be done [18] [36], but often they take for granted the underlying models (for instance the RSD), and offer highly articulated (and most of the times very complex) solutions. Such models often require that the developer focuses much on very complex models, who is required to perform as a human compiler. We think that these complex models as those cited above often indicates a problem related to the language/system [37], and reinforce the necessity of a dedicated language for making video games and our choice of choosing Casanova as the language on which to base our project thesis. Design Goals We summarized the goal that Casanova achieves in the following table. These points describe both the requisites of the RSD model and requirements about expressive power and simplicity of the language.

- Allow a developer to build games without explicit limitations in terms of genre or creativity
- Support common game development tasks of handling the game logic, game AIs, state machines, drawing, and input
- Be concise, in particular when compared with traditional programming languages used for the same Games

- Be efficient, supporting fast run-time execution of the game and common optimizations such as multi-threading or query optimizations

- Be simple to read, supporting syntactic idioms that are grounded in logic and mathematics rather than adopting the syntax of widespread programming languages

- Contain as few features as possible, that is the language should be built on top of few orthogonal [29] constructs that each serve a clear purpose and that mix well together

These objectives make Casanova, a simple language, expressive and efficient, and ideal for making video games and simulations.

4.2 Informal design

Casanova specifies a game by defining the following points:

- The model of the world, that are entities, and values useful to represent and track the game world

- Logic and physics rules in which the objects must obey

- The state machines which manage the not regular dynamic of the game logic, which are not easy to express in term of rules

- The visual aspect of the game entities

- The initial state of the world

For instance, let's see a very simple game written in pseudo-game code which simulates a collection of bouncing balls. We will use this pseudo-code to introduce the idea behind Casanova and in the following section we will see a full Casanova code sample. Our model for the bouncing ball game is defining the world as follow:

```
World = a series of balls
```

A ball then be defined as:

```
Ball =
   The ball position
   The ball velocity
   A sprite of a ball
```

We now define the logical and physics rules of the world and its entities. In our world there is just one blobal rule which yields the valid balls and remove the balls which ore out of the screen:

```
World rules =
   When a ball exits the screen then it is removed from the
      world state
```

The rules of the ball simply deal with its physical attributes and update its appearance:

```
Ball rules =
   The position is the numerical integration of its velocity
   The velocity is the numerical integration of the gravity,
       when the ball touches the groing (screen border) the
      velocity is turned upside-down
   The sprite position is the same as the ball position
```

Thanks to the info just given above the simulation results to be very immediate and it works by running a set of balls and removing those which go out of the screen. Now we add a new behavior which is moving continuously balls in to the screen. We model this by the following script:

```
Wait between one or three seconds
Add to the world a random ball
Repeat
Finally we define the initial world which represents the
   initial state when the simulation is run:
Initial world =
    An empty collection of balls
```

As we can see, the just given description of the ball game is clear and simple to understand, we are not forgetting any aspect of the game definition. Compared to other game systems or languages for developing video games, there is not any mention of game loop neither for the rendering (it is implicit in the ball definition); moreover the creation of a ball is defined a simple process. Also notice that we are not assuming prefabricated physics entities or temporal operations, that is we have described everything ourselves.

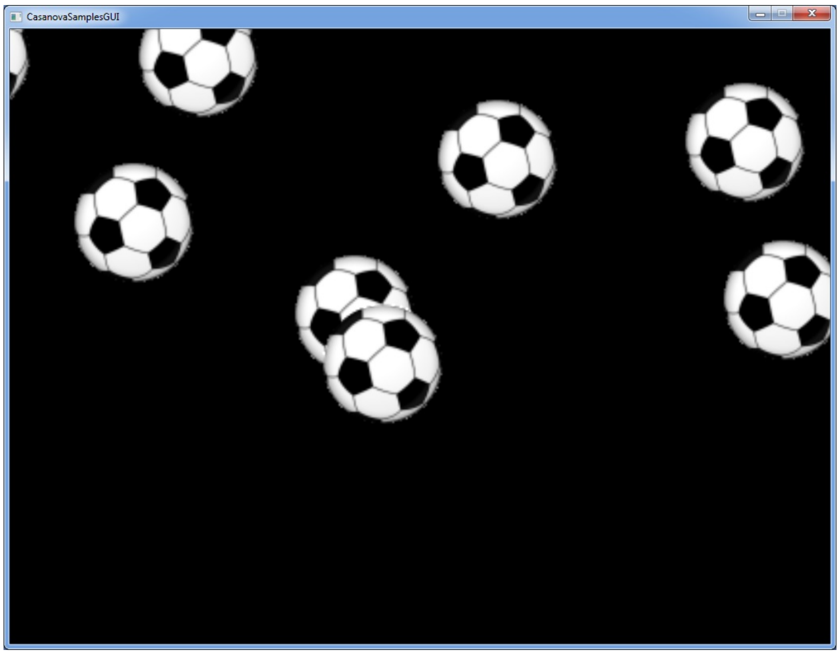

Fig. 4.1: *Casanova bouncing balls in action*

4.3 Casanova in action

Now we show the complete bouncing balls described above using the Casanova language(figure 4.1).

The sample is written with the actual implementation of Casanova, which is based on F# and MonoGame 2.5. It can be used in conjunction with Visual Studio, taking advantage of its powerful tools. Moreover, thanks to MonoGame, it is possible to port Casanova games to multiple platforms, most notably Android smartphones and Tablets or Windows 8 RT. Let us give a look to the sample: a ball simply contains a physics entity and a drawable sprite.

```
type [<CasanovaEntity>] Ball = {
  Circle                : PhysicsEntity
  Sprite                : Sprite
} with
```

The sprite position and rotation are taken directly from the physics.

```
  member self.Position = self.Circle.Position
  static member SpritePosition'(self:Ball,dt:float32<s>)=
    self.Circle.Position
  static member SpriteRotation'(self:Ball,dt:float32<s>)=
    self.Circle.Orientation
```

A wall simply contains a physics entity and a drawable sprite. Ball and Wall are separate entities because extending the sample would likely require adding fields to one or the other. So as a starting point the two entities are almost identical, but in practice it is likely that they would quickly evolve to store different values.

```
and [<CasanovaEntity>] Wall = {
  Box                   : PhysicsEntity
  Sprite                : Sprite
} with
```

Again the sprite position is taken directly from the physics.

```
  static member SpritePosition'(self:Wall,dt:float32<s>) =
    self.Box.Position
```

The game world contains: *i* the physics world (before the physics entities, otherwise updates do not work correctly), *ii* a canvas to draw the entities in a fixed-aspect ratio area of the screen, *iii* the list of balls, *iv* the list of wall.

```
and [<CasanovaWorld>] World = {
  Physics               : PhysicsWorld
  Canvas                : Canvas
  Balls                 : RuleList<Ball>
  Walls                 : List<Wall>
} with
```

Balls are removed when they fall out of the screen.

```
  static member Balls'(world:World,dt:float32<s>) =
    seq{ for b in !world.Balls do
           if b.Position.Y < 600.0f<m> then
             yield b }
```

We start by initializing the physics world and the canvas. The canvas is created under the default sprite layer, as we only draw everything there. The canvas is uniformly stretched, that is it will appear as a square, and it will be smaller than the screen.

```
let rec start_game (start_selector:StartGameSignature<'a>)
                                   (args : StartGameArgs) =

  let physics = PhysicsWorld.Create()
  let canvas = Canvas.Create(args.DefaultLayer,
                             Vector2.Zero,
                             Vector2.One * 1000.0f,
                               StretchMode.Uniform)
```

Then we create a ball with a position (x,y),a radius (r), and a mass of r / 50. Balls move around, so the physics bodies for the balls are dynamic.

```
let mk_ball x y r =
  {
    Circle    = physics.CreateBall(x, y, r, 1.0f<kg> * r/
                                          50.0f<m>,
                                 0.2f, 0.4f, true)
    Sprite    = Sprite.Create(
                  canvas,
                  Vector2<pixel>.Zero,
                  Vector2<pixel>(r * 2.2f<pixel/m>, r *
                                    2.2f<pixel/m>),
                  @"BouncingBalls\ball")
  }
```

We create one static body for the ground and one for the left wall. Walls are still, so the physics bodies for them are static.

```
let ground =
  {
    Box       = physics.CreateBox(0.0f<m>, 495.0f<m>,
                                  500.0f<m>, 10.0f<m>,
                                  0.0f<kg>, 0.0f, 0.4f,
                                  false)
    Sprite    = Sprite.Create(
                  canvas,
                  Vector2<pixel>.Zero,
                  Vector2<pixel>(1000.f<pixel>,
                                       10.0f<pixel>),
                  0.0f<rad>,
                  Vector2<pixel>.Zero,
                  @"BouncingBalls\wall",
                  Color.White,
                  true)
  }
let left_wall =
  {
    Box       = physics.CreateBox(-500.0f<m>, 0.0f<m>,
                                  10.0f<m>, 500.0f<m>,
                                  0.0f<kg>, 0.0f, 0.4f,
                                  false)
    Sprite    = Sprite.Create(
                  canvas,
                  Vector2<pixel>.Zero,
                  Vector2<pixel>(10.f<pixel>,
                                       1000.0f<pixel>),
                  0.0f<rad>,
                  Vector2<pixel>.Zero,
                  @"BouncingBalls\wall",
                  Color.White,
                  true)
  }
```

The game world starts with: *i*the physics engine,*ii*the canvas,*iii*no balls,*iv*the two walls.

```
  let world =
    {
      Physics = physics
      Canvas  = canvas
      Balls   = RuleList.Empty
      Walls   = [ground; left_wall]
    }
```

We add balls in two different scripts(main and input), so we create an auxiliary script for this purpose. *add-ball* creates a ball in a random position above the screeen and with a random radius between 25 and 100m.

```
  let add_ball =
    co{
      do world.Balls.Add(mk_ball (random_range -400.0f<m>
                                              400.0f<m>)
                                  -650.0f<m>
                                  (random_range 25.0f<m>
                                                100.0f<m>))
      do! yield_
    }
```

The main script adds a random ball every second.

```
  let main =
    repeat_
     (co{
        do! add_ball
        do! wait 1.0f<s>
      })
```

Five input scripts: *i* when the user presses the F9 key the game will be saved in BouncingBalls file;*ii*when the user presses the F10 key the game will be loaded from BouncingBalls file;*iii*when the user presses space a new ball is created *iv*when the user presses backspace a ball (if there are any) is removed *v*when the user presses the escape key the pause menu will be pushed on the stack and displayed.

```
let input =
  [
    wait_key_press Keys.F9  => args.Save("BouncingBalls");
    wait_key_press Keys.F10= > args.Load("BouncingBalls");
    wait_key_down Keys.Space =>
      co{
        do! add_ball
        do! wait_key_up Keys.Space .||> wait 0.2f<s>
      }
    wait_key_down Keys.Back =>
      co{
        if world.Balls.Count > 0 then
          do world.Balls := world.Balls |> Seq.skip 1
        do! wait_key_up Keys.Back .||> wait 0.2f<s>
      }
    wait_key_press Keys.Escape
              => args.PushStack(ConfirmationMenu.
                 start_menu
                                   (args.SetStack
                                      start_selector),
                                   false)
  ]
world, main, input
```

Chapter 5

The REA design pattern

In this section we will define a model for an algebra tho show that the REA (Resource Entity Action) model can be reduced to a problem of linear algebra. We then show how games that use this model can be further simplified by linguistic constructs. In an RTS game we can think of the game world as a collection of entities. Entities perform an action by exchanging resources with one another, thus the resources may be stored in a vector and resource exchange may be expressed by matrix algebra.

5.1 Action algebra

As described in Section 2, we can consider an action as a flow of resources from a source entity to one or more target entities. We require that each entity has a resource vector, which contains the current amount of resources of the entity. The resource vector is sparse since most actions involve only few resource types. An action is expressed by a transformation matrix A which determines how a resource is passed to another entity for that particular action.

Let us consider a set of target entities $T = \{t_1, t_2, ..., t_n\}$ which are the targets of the action and a source entity e. Each entity t_i (including the source entity type) owns a resource vector $\mathbf{r_i} = (r_{i_1}, r_{i_2}, ..., r_{i_m})$ while the source entity owns also a transformation matrix A of size $m \times m$, which defines how the h-th component (i.e., resource) of the resource vector is affected by the interaction with the k-th resource. We also consider an integrator dt which contains the time difference between the current frame and the previous one. We then compute $\mathbf{w_e} = (w_{e_1}, w_{e_2}, ..., w_{e_m}) = \mathbf{r_s} \times A \cdot dt$. From the definition of matrix multiplication, it immediately follows that each component of $\mathbf{w_e}$ represents how the h-th resource will change by applying the effect of all the other resources to it. Now we compute the vector $\mathbf{r'_i} = \mathbf{r_i} + \mathbf{w_e}\ \forall e_i \in E$ which will now replace the resource vector in each target entity.

For instance, consider the action of a spaceship entity using laser to damage (resource) an enemy spaceship (entity). This involves a vector resource of two elements: laser and life points. The action must transfer laser points to subtract from the enemy life points. Let us assume the vector resource of the targeting ship is $r_s = (20, 500)$ and the vector resource of the targeted ship is $r_t = (15, 1000)$. Let the transformation matrix be

$A = \begin{bmatrix} 0 & -1 \\ 0 & 0 \end{bmatrix}$ which means that the source entity will affect the life of the target with a negative number of laser points. Thus $w_e = r_s \times A \cdot dt = (20, 500) \times A \cdot dt = (0, -20) \cdot dt$. At this point, assuming $dt = 1$ second, we have $r'_t = r_t + w_e = (20, 1000) + (0, -20) \cdot dt = (20, 980)$.

5.2 A declarative language extension

The REA design pattern is modeled using the action algebra. We will now describe a language extension that implements this design pattern/algebra for the Casanova game programming language. The language extension is purely declarative. Its semantics are described using the SQL query language, which has the advantage of familiarity to most programmers.

Implementing the action algebra may be done using an abstract class which contains an abstract method which performs the action. Each action is a class which extends the previous abstract class and implements the abstract method. This method will fetch the world looking for the information needed to find what entities are affected by the action execution. Each entity of the game will have a collection of actions it can perform, automatically run by Casanova.

We still lack a way to identify the set of target entities T given a source entity and its action. A solution that lets the programmer implement class inheritance and abstract methods will produce a lot of boilerplate code, where the common behavior is scanning the game world in search of entities satisfying certain properties needed to apply the effect of an action. The reason is that the library lacks knowledge of the game definition, which must be compensated for by the game programmer.

To avoid boilerplate code to a great extent, we can try to hide repeated patterns, by capturing it as a language construct with its syntax and semantic extending Casanova. Such an extension allows us to alleviate the problem of finding entities because, at compile-time, we can explore the shape of the game world and generate the appropriate world exploration code. To do so we add a new type definition: the *action*. An action is a declarative construct which is used to describe not only the resource exchange between entities, but also what kinds of entities participate in the exchange. The resource exchange is based on *transfers* (Add, Subtract, and Set), while the target determination is based on *predicates*: we filter the game world entities depending on their types,attributes and radius (specifying the distance beyond which the action is not applied). Some actions, called threshold actions, are not continuous and make use of special predicates to delay the execution (Output) until certain conditions are met.

Using actions it is possible to specify an exchange of resources in a fully declarative manner, so that the developer does not have to rewrite similar pieces of code ad hoc for each action.

Chapter 6

Syntax and semantic of REA

We now give the syntax and semantics of actions in Casanova. The grammar allows the definition of actions, which make up the body of spacial Casanova entities which act as placeholders for actions. When an entity contains such an action, the Casanova runtime will apply it to all appropriate targets.

6.1 Action Grammar Definition

In this section we will define formally the grammar definition for actions. To better clarify the use of this grammar, we will first provide a taxonomy for our actions. We divide our actions into three kinds: (1) constant transfer actions, (2) mutable transfer actions, and (3) threshold actions.

Constant Transfer:

Constant transfer actions update the target fields with a constant value or a value taken from one of the source fields. The source field is not affected by the resource transfer. An example of a constant transfer action is a defense tower shooting an arrow at an infantry unit.

```
TARGET Infantry; RESTRICTION Owner <> Owner; RADIUS 1000.0;
   TRANSFER CONSTANT Life - ArrowDamage;
```

Mutable Transfer:

Mutable transfer actions are used when the resource exchange affects not only the target, but also the source entity. The source field is updated depending on the used operation: if we add a resource to the target, then the same amount is subtracted to the source field, if we subtract a resource then the same amount will be added while if we set the target to a value the source is not changed. An example of a mutable transfer action is a spaceship transferring minerals from its holds to a shipyard.

```
TARGET Shipyard; RESTRICTION Owner = Owner; RADIUS 150.0;
   TRANSFER MineralStash + Minerals;
```

Threshold Action:

Threshold actions follows the same transfer semantics of the previous two types of actions. In addition, they have a collection of threshold values and output operations. The output operations are executed once when all the threshold values are reached. The threshold values are on fields belonging to the source entity. The output operations modify only fields of the source entity following the same semantic of the transfer operations. An example for a threshold action is a worker building a town hall. When the `integrity` of the town hall reaches 100, a flag `completed` is set (which is one of its fields) which warns the system to replace the partial constructed building with the complete building.

```
TARGET ConstructionTownHall; RESTRICTION Owner = Owner;
   RADIUS 10.0; TRANSFER CONSTANT Integrity + 1.0; THRESHOLD
    Integrity = 100.0; OUTPUT Completed := true
```

Now we give a formal definition for the grammar instances presented in the examples above. In the following formal grammar definition we used reference to Casanova types: Casanova Entity is an entity in the game world represented as a record with its fields; the special keyword `Self` is used to point the entity owning the action as one if its fields. To define the grammar we use the extended Backus-Naur form.

```
<Action> ::= TARGET <TARGET LIST> <RESTRICTION LIST> [<
   RADIUS CLAUSE>] <TRANSFER LIST>
   <INSERT LIST> [<THRESHOLD BLOCK>]
<TARGET LIST> ::= <ACTION ELEMENT>+
<ACTION ELEMENT> ::= Casanova Entity | Self
<RESTRICTION LIST> ::= {<RESTRICTION CLAUSE>}
<RESTRICTION CLAUSE> ::= RESTRICTION Boolean Expression of <
   SIMPLE PRED>
<SIMPLE PRED> ::= Self Casanova Entity Field (= | <>) Target
    Casanova Entity Field
<TRANSFER LIST> ::= {<TRANSFER CLAUSE>}
<TRANSFER CLAUSE> ::= (TRANSFER | TRANSFER CONSTANT)
(Target Casanova Entity Field) <Operator> ((Self Casanova
   Entity Field) | (Field Val)) [* Float Val]
<Operator> ::= + | - | :=
<RADIUS CLAUSE> ::= RADIUS (Float Val)
<INSERT LIST> ::= {<INSERT CLAUSE>}
```

```
<INSERT CLAUSE> ::= INSERT (Target Casanova Entity Field) ->
    (Self Casanova Entity Field List)
<THRESHOLD BLOCK> ::= <THRESHOLD CLAUSE>+
<OUTPUT CLAUSE>+
<THRESHOLD CLAUSE> ::= THRESHOLD
(Self Casanova Entity Field) Field Val
<OUTPUT CLAUSE> ::= OUTPUT
(Self Casanova Entity Field) <Operator> ((Self Casanova
   Entity Field) | (Field Val)) [* Float Val]
```

6.2 Formal semantic definition

Given the fact that our actions resemble queries on entities, we specify their semantics as translation semantics to SQL. This allows us to leverage existing discussions on SQL correctness [39].

In defining our translation rules formally, we consider a set $T = \{t_1, t_2, ..., t_n\}$ of target types and a source entity type s. In all actions we select a subset of targets in each t_i, on which applying the action, using restriction conditions (if any exist, otherwise the targets of type t_i are used). After that we apply the resource transfer (which, as explained above, can be either from a constant field or a mutable field).

We assume that each entity type is represented by an SQL relation and that there exists a key attribute called **Id** for each relation. We now consider each of the three translation cases, based on the taxonomy given above, separately. In the following translation rules we will use, for greater clarity, notations inside the SQL code taken from the Backus-Naur form for grammar definitions such as [expr] to denote an optional expression or "|" to denote a choice between expressions. We also extend the SQL grammar with a global variable *dt* which is the time difference between the current and the last game frame. In this way the increment of the entity attribute values are proportional to the elapsed time. All types of actions evaluate the predicates in the restriction conditions and apply a filter to their targets. All targets further than the radius are automatically discarded when executing the action. The transfer predicates are executed immediately on all filtered targets.

CONSTANT TRANSFER:

In constant transfers we must update each target t_i satisfying the restriction conditions with the value in the source fields or constant values specified in the transfer clause. For simplicity, we will assume that constant values will be stored as attributes of the source entity.

Let us consider a set of resource attributes $A = \{a_{j_1}, a_{j_2}, ..., a_{j_m}\}$ of the source entity used to update the target t_i. We have to compute the contribution of all sources of the same type on the target t_i. We want to produce a relation whose tuples represent the target id followed by the total amount of resource a_{j_r} to transfer, called Σ_r:

Transfer				
ID	Σ_1	Σ_2	$\cdots$	Σ_m

The following SQL instruction implements the relation definition above:

```
SELECT  $t_i$.id, SUM($s.a_{j1}$} AS $\Sigma_1$,
        SUM($s.a_{j2}$} AS $\Sigma_2$,...,
        SUM($s.a_{jm}$} AS $\Sigma_m$

FROM    Target $t_i$, Source $s$
WHERE   <RESTRICTION LIST> [AND <RADIUS CLAUSE>]
GROUP BY $t_i.id$
```

Now $\forall t_i \in T$ we must update the target attributes $A' = \{a_{t_1}, a_{t_2}, ..., a_{t_m}\}$ using one of the target operators defined in the grammar (Set, Add, Subtract) with the attributes of the previous relation scheme.

```
WITH    Transfer AS(
                SELECT  $t_i$.id, SUM($s.a_{j1}$} AS $\Sigma_1$,
                        SUM($s.a_{j2}$} AS $\Sigma_2$,...,
                        SUM($s.a_{jm}$} AS $\Sigma_m$)

FROM    Target $t_i$, Source $s$
WHERE   [<RESTRICTION LIST>] [AND <RADIUS CLAUSE>]
GROUP BY $t_i.id$)
UPDATE  Target $t_i$
SET     $t_i.a_{t_1}$ = u.$\Sigma_1$ | $t_i.a_{t_1}$ = $t_i.a_{t_1}$ + u.$\Sigma_1$ * $dt$ | $t_i.a_{t_1}$ =
$t_i.a_{t_1}$ - u.$\Sigma_1$ * $dt$\
...
FROM    Transfer $u$
WHERE   $u.id = t_i.id$
```

MUTABLE TRANSFER:

In a mutable transfer the field of the source involved in the resource transfer must be updated depending on the applied transfer operator. If the operator is Add then the resource is subtracted from the source field and added to the target field proportionally do *dt*. If the operator is Subtract then the resource is subtracted from the target field and added to the value of the source field proportionally to *dt*.

The first step in translating this semantic rule is finding how many targets (if any) are affected by each source entity, in order to obtain the following relation scheme:

TotalTargets	
Source ID	*TargetCount*

The SQL code implementing the previous scheme is the following:

```
TotalTargets =
SELECT  s.id,COUNT(*) AS TargetCount
FROM    Source s, Target t1, Target t2,...,Target tn
WHERE   <RESTRICTION LIST> [AND <RADIUS CLAUSE>]
GROUP BY s.id
HAVING  COUNT(*) > 0
```

Now $\forall t_i \in T$ we need to obtain a relation storing what target each of the source entity is affecting and the count of affected targets. The following relation scheme describes what we have just informally explained:

OutputSharing		
Source ID	*Target ID*	*Output Sharing*

For brevity in the code below it has been used the notation RelationName = [...] which means the code for that table has been previously defined. The following SQL code implements the previous scheme:

```
OutputSharing =
SELECT  *
FROM    TotalTargets c, SourceOutput c1
WHERE   c.s_id = c1.s_id
        AND SourceOutput =
                SELECT  s.id AS s_id,ti.id AS t_id
                FROM    Source s, Target ti
                WHERE   <RESTRICTION LIST> [AND <RADIUS
                   CLAUSE>]
        AND TotalTargets = [...]
```

Each target attribute receives an amount of resources equal to the total transferred resources divided by the number of targets receiving that resource from the source. The latter value can be read in the table obtained with the SQL code above. Formally, let $a_{j_k} \in A$ be the resource attribute containing the total amount to be transferred and c the number of targets affected by the resource transfer, then each target receives $R_k = a_{j_k}/c$ resources from each source. The values needed to compute R_k can be obtained both from *OutputSharing* and the Source relation. The complete SQL code to update the target t_i is the following:

```
WITH     Transfer AS(
         SELECT  $t_i$.id, SUM($s.a_{j_1}$ / o.TargetCount) AS $\Sigma_0$,SUM($s.a_{j_2}$
             / o.TargetCount) AS $\Sigma_2$,...,SUM($s.a_{j_m}$ / o.
            TargetCount) AS $\Sigma_m$
         FROM    Source s, Target $t_i$,OutputSharing o
         WHERE   OutputSharing = [...] AND s.id = o.s_id AND
            t.id = o.t_id)
         GROUP BY $t_i$.id
UPDATE   Target $t_i$
SET      $t_i.a_{t_1}$ = u.$\Sigma_1$ |$t_i.a_{t_1}$ = $t_i.a_{t_1}$ + u.$\Sigma_1$ * $dt$ |$t_i.a_{t_1}$ = $t_i.a_{t_1}$ - u.$\Sigma_1$
    * $dt$
...
FROM     Transfer u
WHERE    $t_i$.id = u.id
```

To update the Source relation we use a relation similar to the one use to update the target, but this time there is no need to save the count of the affected targets but just to check if the source has at least one target, otherwise it is not updated.

```
WITH     TotalTransfer AS(
         SELECT  s.id,s.$a_{j_1}$,s.$a_{j_2}$,...,s.$a_{j_m}$
         FROM    Source s, Target $t_1$,...,Target $t_n$
         WHERE   <RESTRICTION LIST>
                 [AND <RADIUS CLAUSE>]
         GROUP BY        s.id,$s.a_{j_1}$,$s.a_{j_2}$,...,$s.a_{j_m}$
         HAVING  COUNT(*) > 0)

UPDATE   Source s
SET      s.$a_{j_1}$ = s.$a_{j_1}$ - s.$a_{j_1} * dt$|s.$a_{j_1}$ = s.$a_{j_1}$ + s.$a_{j_1} * dt$
...
FROM     TotalTansfer u
WHERE    s.id = u.id
```

THRESHOLD TRANSFER:

A threshold action is an action defined as the previous two types, i.e. it has a resource transfer definition which is always executed, and a set of threshold conditions that, if met, activate the Output operations, which are always towards the source entity.

The attributes of the source entity affected by Output operations can be updated with constant values or values from other attributes in the source entity. In the latter case the transfer is treated as in the mutable transfer case.

Let us consider a set of updating attributes $U = \{a_{k_1}, a_{k_2}, ..., a_{k_l}\}$ and a set of attributes to be updated $U' = \{a_{s_1}, a_{s_2}, ..., a_{s_l}\}$ in the output operation. We must first check that all the conditions in the threshold clauses are met, then we have to update the attributes in the source entity appropriately.

```
WITH     TotalOutput AS(
         SELECT  s.id,s.$a_{k_1}$,s.$a_{k_2}$,...,s.$a_{k_l}$
         FROM    Source s
         WHERE   <THRESHOLD CLAUSE 1>
                 [AND <THRESHOLD CLAUSE 2>]
                 .
                 .
                 .
                 [AND <THRESHOLD CLAUSE $l$>])
UPDATE   Source s
SET      s.$a_{s_1}$ = o.$a_{k_1}$|s.$a_{s_1}$ = (s.$a_{s_1}$ + o.$a_{k_1}$)$*dt$; o.$a_{k_1}$ = o.$a_{k_1}$ - o
   .$a_{k_1}*dt$|s.$a_{s_1}$ = (s.$a_{s_1}$ - o.$a_{k_1}$)$*dt$; o.$a_{k_1}$ = o.$a_{k_1}$ + o.$a_{k_1}*dt$
...
FROM     TotalOutput o
WHERE    s.id = o.id
```

6.3 Casanova implementation

The process of evaluating actions was added to Casanova, which, using a compiler, generates assembly code specific for each action. The generated code executes the actions at each game frame. Besides, the compiler checks that the targets are valid and that the fields used in all the predicates are contained in those entities. To improve performance an index is built at compile time, to speed up resolution of radius restrictions. The implementation uses type attributes for actions, so the syntax is different even though there is a mapping between elements of the syntax presented here and those of the concrete syntax.

Chapter 7

Case study

We now present an RTS game we used as a case study and the benchmarks that test the action implementation. We call this game "CS" for "case study." In the game players must conquer a star system made up of various planets. Each planet builds fleets which are used to fight the fleets of the other players and to conquer more planets. A planet is conquered when a fleet of a player is near it and no other enemy fleet is defending it.

7.1 Case study

Three actions are required in this game: The first action, called `Fight Action`, defines how a fleet fights enemy fleets in range. The fight action subtracts $0.5 \cdot dt$ `life` points to the enemy fleet during every action tick (every frame) which are in range.

```
Fleet = {Position: Rule Vector2;FightAction: FightAction;
   Owner: Ref Player;Life: Var float32;Fight: FightAction }
```

The fight action is defined as follows:

```
FightAction = TARGET Fleet; RESTRICTION Owner <> Owner;
   RADIUS 150.0; TRANSFER CONSTANT Life - 0.5;
```

The action target is an entity whose type is **Fleet**, the condition to execute the action is that the fleet must be an enemy, so the fleet owner must be different as specified in the `Restriction` clause. The `attack range` is 150 units of distance, so the `Radius` will be 150. When atacked 0.5 `life` points are subtracted at every attack.

The second action is called `BuildAction` and allows a planet to create a ship. In order to build a ship, a planet must gather 10 mineral units. Each planet has a field called `GatherSpeed` which determines how fast it gathers minerals. Every tick the planet mineral stash is increased by this amount. This action is a threshold action where the threshold value is the minerals of the planet. As soon as the threshold value is reached, we set the field `NewFleet` to TRUE (it is used by the engine to create a new fleet) and `Minerals` to 0 to reset the counter. The planet and its actions are:

```
Planet = {Position: Vector2;Owner: Rule Ref Player;NewFleet:
     Rule bool;BuildAction:BuildAction;
   EnemyOrbitingFleetsAction : EnemyOrbitingFleetsAction;
   GatherSpeed: float32;Minerals: Var float32 }
```

```
BuildAction =
TARGET Self; TRANSFER CONSTANT Minerals + GatherSpeed;
   THRESHOLD Minerals 10.0; OUTPUT NewFleet := true; OUTPUT
   Minerals := 0.0
```

A Casanova rule is appointed to read the value of NewFleet and, if it is true, it spawns a new fleet.

The third action is required to check if a planet can be conquered by a fleet. A fleet can conquer a planet if there is no enemy fleet near it and if it is close enough. Thus the action definition will be the following:

```
EnemyOrbitingFleetsAction =
TARGET Fleet; RESTRICTION Owner Not Eq Owner; RADIUS 25.0;
   INSERT Owner -> EnemyOrbitingFleets
```

The action will add an enemy fleet close enough in a data structure used by a Casanova rule to change the owner of the planet.

Even the concept of drawing lasers can be implemented using the INSERT clause simply adding it to `FightAction` which inserts in a list all the targeted ships positions. In this way we can draw a laser from the source position to the target position. We omit this aspect for brevity.

7.2 REA in action

Below we show a complete example of an RTS game written with the actual implementation of REA in Casanova. We omit the initial world and the world definition because they do not use such pattern and are useless to our demonstration. The sample actually run and can be downloaded at the Casanova website together with others samples which use REA as a way to define resources exchange between entities.

The star system is a CasanovaWorld, in that it acts as a local root for all the entities it contains. Those entities do not know anything about the game world and can only interact with the other entities inside the star system itself. The star system contains:(*i*)a reference to the drawable layers; it is important that layers, inside the star system act as a reference, otherwise everything will be rendered twice, (*ii*) a list of planets, (*iii*) a list of fleets, (*iv*) a list of references to players.

```
type [<CasanovaWorld>] StarSystem =
  {
    Layers        : Ref<Layers>
    Planets       : List<Planet>
    Fleets        : RuleList<Fleet>
    Players       : Ref<Player>[]
  } with
```

We create the star system from the players and the layers. First of all we convert all players to references then we initialize a randomized grid of planets from a list of possible planet textures and sizes.Bigger planets build ships faster.

```

  static member Create(players:Player[], layers) =

    let players = [| for p in players do yield ref p |]
    let mutable planets = []
    for i = -10 to 10 do
      for j = -10 to 10 do
        if random_interval 0 10 <= 3 then
          let texture,size =
            [
              @"RTS\Planets\earth", 50.f
              @"RTS\Planets\findolfin.png", 30.f
              @"RTS\Planets\fungo.png", 40.f
              @"RTS\Planets\ice.png", 50.f
              @"RTS\Planets\marte.png", 40.f
              @"RTS\Planets\pluto.png", 30.f
              @"RTS\Planets\saturn.png", 70.f
              @"RTS\Planets\thor.png", 80.f
            ] |> List.RandomElement
          let new_planet = Planet.Create(Vector2.Create(i, j
             ) * 120.0f + Vector2.Create(random_range -25.0f
              25.0f, random_range -25.0f 25.0f) * 0.25f,
             layers, texture, size, players.[ (
             random_interval 0 players.Length) ])
          do planets <- new_planet :: planets
    {
      Players       = players
      Layers        = ref layers
      Planets       = planets
      Fleets        = RuleList.Create (fun () -> Seq.empty)
    }
```

The fleets list is updated by keeping all the fleets that are still alive, plus all the fleets from planets that just finised building; when a planet is done building it will set, for just one frame, the NewFleet field to true. Notice that we do not build more than a given number of fleets in total.

```
  static member FleetsRule(self : StarSystem) =
    seq{
      for p in self.Planets do
        if !p.NewFleet && self.Fleets.Value |> Seq.length <
           300 then
          yield Fleet.CreateFrom(!(!p.Owner), 1.0f, p.
             Position, @"RTS\Ships\ship", !self.Layers)
      for f in self.Fleets do
        if !f.Life > 0.0f then
          yield f }
```

Fleets implement a large portion of the logic of the game. A fleet handles basic flocking and path-finding, maintains information on its owner, handles battles, and draws its texture, selection halo, lasers, and health bar. Position, target, velocity and max speed determine movement for the ship. Neighboring fleets contains all the nearby fleets in order to avoid excessive superposition. The list is filled by the FindNeighboringFleet-sAction, which performs spatial optimizations in order to find the neighbors quickly. Fighting involves the current health of the ship and the fight action which takes care of modifying the life of ships when enemies enter in range. The fight action also fills the laser targets with all the fleets that are currently being attacked; these fleets are then linked to the current one by lines that represent lasers. The ship draws its own texture, a shadow that shows the color of the owner and the current selection, and a line that represents the fleet's health.

```
and [<CasanovaEntity>] Fleet =
  {
    Position     : Rule<Vector2<m>>
    Target       : Var<Vector2<m>>
    Velocity     : Rule<Vector2<m/s>>
    MaxSpeed     : Var<float32<m/s>>
    NeighbouringFleets : RuleList<Vector2<m>>
    FindNeighbouringFleetsAction :
       FindNeighbouringFleetsAction
    Owner        : Ref<Player>
    Life           : Var<float32>
    Fight          : FightAction
    LaserTargets   : RuleList<Ref<Fleet>>
    Lasers         : RuleList<Line>
    Appearance   : Sprite
    FleetShadow  : Sprite
    Stats          : Line
    Selected       : Var<bool>
  } with
```

LasersRule: create one line for each laser target from self to the target. *this.SetTarget*: when a new target is sent to this fleet, then zero the velocity and set the target. *StatsSourceRule*: the health bar moves with the fleet. *StatsDestRule*: the length of the health bar reflects the health; longer = healthier. *StatsColorRule* the color of the health bar reflects the health; greener = healthier. *AppearancePositionRule*: the texture moves with the fleet. *FleetShadowPositionRule*: the shadow moves with the fleet. *FleetShadowScaleRule*: the shadow is bigger and darker when the ship is selected. *NeighbouringFleetsRule*: the velocity moves towards the target, but it also moves away from the neighboring fleets. The neighboring fleets are filled by the FindNeighbouringFleetsAction action with a spatial partitioning optimization. *PositionRule*: the position of a fleet is updated by its velocity, unless it reached its target.

```
  static member LaserTargetsRule(self:Fleet) : seq<Ref<Fleet
     >> = Seq.empty
  static member LasersRule(world:StarSystem, self:Fleet) =
    seq{
      for t in self.LaserTargets do
        let t = !t.Value.Position
        yield Line.Create((!world.Layers).Overlay, !self.
           Position * 1.0f<_>, t * 1.0f<_>, @"UI\laser",
           Color.White, 1.0f)
    }
  member inline this.SetTarget(target : Vector2<m>) =
    this.Velocity := Vector2.Create(0.0f)
    this.Target := target
  static member StatsSourceRule(self:Fleet) = !self.Position
     + Vector2.UnitY * 10.0f - Vector2.UnitX * 5.0f
  static member StatsDestRule(self:Fleet) = !self.Position +
     Vector2.UnitX * !self.Life + Vector2.UnitY * 10.0f -
    Vector2.UnitX * 5.0f
  [<RuleUpdateFrequency(UpdateFrequency.AI)>]
  static member StatsColorRule(self:Fleet) = Color.Lerp(
    Color.Red, Color.Green, !self.Life / 10.0f)
  static member AppearancePositionRule(self:Fleet) = !self.
    Position
  static member FleetShadowPositionRule(self:Fleet) = !self.
    Position
  [<RuleUpdateFrequency(UpdateFrequency.Interactive)>]
  static member FleetShadowScaleRule(self:Fleet) =
    if !self.Selected then Vector2 1.25f else Vector2 1.0f
```

```
[<RuleUpdateFrequency(UpdateFrequency.Interactive)>]
static member FleetShadowColorRule(self:Fleet) =
  if !self.Selected then (!self.Owner).Color else Color.
     Lerp((!self.Owner).Color, Color.White, 0.25f)
static member NeighbouringFleetsRule(self:Fleet) : seq<
   Vector2<m>> = Seq.empty
[<RuleUpdateFrequency(UpdateFrequency.AI)>]
static member VelocityRule(star_system:StarSystem,self:
   Fleet,dt:float32<s>) =
  let target_distance = Vector2<m>.Distance(!self.Position
     ,!self.Target)
  let repulsions =
    seq{
      for f' in self.NeighbouringFleets do
        let strength = smoothstep(Vector2.Distance(f', !
           self.Position) / 50.0f<m>) 5.0f 0.0f
        let away =
          if Vector2.Distance(!self.Position, f') <= 0.1f<
             m> then Vector2.Create(random_interval -1 1,
             random_interval -1 1) * 5.0f
          else Vector2.Normalize(!self.Position - f')
        yield away * strength
    }
  let repulsions = repulsions |> Seq.fold (+) Vector2.Zero
  let try_normalize (v:Vector2<1>) =
    if v.Length <= 0.1f then v
    else v |> Vector2.Normalize
  let velocity'=
    if target_distance < 0.1f<m> then
      (repulsions |> try_normalize) * !self.MaxSpeed
    else
      (!self.Target - !self.Position).Normalized * !self.
        MaxSpeed + (repulsions |> try_normalize) * !self.
        MaxSpeed
  Vector2.Lerp(!self.Velocity, velocity', dt * 1.0f<_>)
static member PositionRule(star_system:StarSystem,self:
   Fleet,dt:float32<s>) =
  let target_distance = Vector2<m>.Distance(!self.Position
     ,!self.Target)
  let new_position = !self.Position + dt * !self.Velocity
  if target_distance < 0.1f<m>  then
    !self.Target
  else
```

```
      new_position
```

We create a new fleet with empty counters, loaded textures, and full life.

```
  static member CreateFrom(owner : Player, laser,
     planet_position : Vector2<m>, texture, layers : Layers)
      : Fleet =
    let pos = planet_position - Vector2<m>.UnitY * 70.0f
    {
      LaserTargets        = RuleList.Create(fun () -> Seq.
         empty)
      NeighbouringFleets  = RuleList.Create(fun () -> Seq.
         empty)
      Lasers              = RuleList.Create(fun () -> Seq.
         empty)
      Position    = Rule.Create(planet_position + Vector2<m
         >.Create(random_range -1.0f 1.0f, random_range -1.0
         f 1.0f) * 10.0f)
      FleetShadow = Sprite.Create(layers.Planets,
         planet_position * Vector2.One, Vector2<pixel>.One *
          70.f, @"UI\selection_frame", owner.Color)
      Target      = var(planet_position + Vector2.Create(
         random_range -1.0f 1.0f, random_range -1.0f 1.0f) *
          50.0f)
      Velocity    = Rule.Create(Vector2.Zero)
      MaxSpeed    = var 50.0f<m/s>
      Owner       = ref owner
      Life        = var 10.0f
      Stats       = Line.Create(layers.Overlay,
         planet_position  * 1.0f<_>, planet_position  * 1.0f
         <_>, @"UI\white_pixel", Color.White, 5.0f)
      Appearance  = Sprite.Create(layers.Fleets,
         planet_position * Vector2.One, Vector2<pixel>.One *
          50.f, texture)
      Selected    = var false
      Fight       = FightAction
      FindNeighbouringFleetsAction =
         FindNeighbouringFleetsAction
    }
```

Planets mainly handle building of new fleets. When an enemy fleets are the closest to the planet, then it is conquered and it changes owner. Building of fleets is done through the BuildAction action which accumulates minerals. When the target amount of minerals is reached, then NewFleet is set to true for one frame. The planet also stores a position, an owner, and its visuals. The visuals are made up of a texture for the planet, a bar that signals building completion, and a halo that determines the planet owner.

```
and [<CasanovaEntity>] Planet =
  {
    Position        : Vector2<m>
    Owner           : Rule<Ref<Player>>
    Appearance      : Sprite
    PlanetShadow    : Sprite
    Stats           : Line
    NewFleet        : Rule<bool>
    BuildAction     : BuildAction
    GatherSpeed     : float32
    Minerals        : Var<float32>
    EnemyOrbitingFleets : RuleTable<Ref<Player>>
    EnemyOrbitingFleetsAction : EnemyOrbitingFleetsAction
  } with
```

PlanetShadowColorRule: the color of the background halo of the planet is the same as the planet owner. It is updated with low frequency. *StatsDestRule*: the stats bar length depends on the amount of gathered minerals. It is updated with low frequency. *StatsColorRule*: the stats bar color depends on the amount of gathered minerals. It is updated with low frequency. *EnemyOrbitingFleetsRule*: the owner is changed if only enemy fleets are orbiting the planet. It is updated with low frequency. *NewFleetRule*: is reset to false after every tick, but it is set to true by the BuildAction action.

```
  [<RuleUpdateFrequency(UpdateFrequency.AI)>]
  static member PlanetShadowColorRule(self:Planet) = (!(!
     self.Owner)).Color
  [<RuleUpdateFrequency(UpdateFrequency.AI)>]
  static member StatsDestRule(self:Planet) = self.Position +
      Vector2.UnitX * !self.Minerals * 2.0f + Vector2.UnitY
     * 10.0f - Vector2.UnitX * 10.0f
  [<RuleUpdateFrequency(UpdateFrequency.AI)>]
  static member StatsColorRule(self:Planet) = Color.Lerp(
     Color.CornflowerBlue, Color.GreenYellow, !self.Minerals
      / 10.0f)
  static member EnemyOrbitingFleetsRule(self : Planet,
     star_system : StarSystem) : seq<Ref<Player>> = Seq.
     empty
  [<RuleUpdateFrequency(UpdateFrequency.AI)>]
  static member OwnerRule(self : Planet, star_system :
     StarSystem) =
    if Seq.isEmpty self.EnemyOrbitingFleets then
      !self.Owner
    else
      Seq.head self.EnemyOrbitingFleets
  static member NewFleetRule(self : Planet, star_system :
     StarSystem) = false
```

We create a new planet with empty counters and loaded visuals.

```
  static member Create(position, layers, texture, size :
     float32, owner : Ref<Player>) =
    {
      EnemyOrbitingFleetsAction = EnemyOrbitingFleetsAction
      EnemyOrbitingFleets = RuleTable.Create(fun () -> Seq.
         empty)
      Position       = position
      Appearance     = Sprite.Create(layers.Planets, position
          * Vector2.One, Vector2<pixel>.One * size, texture)
      PlanetShadow   = Sprite.Create(layers.Planets, position
          * Vector2.One, Vector2<pixel>.One * size * 1.5f<_
         >, @"UI\selection_frame")
      Stats          = Line.Create(layers.Overlay, position *
          1.0f<pixel/m> + Vector2.UnitY * 10.0f - Vector2.
         UnitX * 10.0f, position * 1.0f<pixel/m>, @"UI\
         white_pixel", Color.White, 5.0f)
      Owner          = Rule.Create(owner)
      NewFleet       = Rule.Create false
      BuildAction    = BuildAction
      GatherSpeed    = size / 50.0f
      Minerals       = var 0.0f
    }
```

The player simply contains a name and a color.

```
and [<CasanovaEntity; ReferenceEquality>] Player = {
  Name           : string
  Color          : Color
} with static member Create(name, color) = { Name = name;
   Color = color}
```

The EnemyOrbitingFleetsAction selects the enemy fleets which are orbiting within a ray of 25 around the planet.

```
and [<Action.Action(Target = "Fleet");
       Action.Restrict(Condition = Action.Condition.NotEqual
           , Field = "Owner");
```

```
        Action.Insert(Value = "Owner", To = "
           EnemyOrbitingFleets");
        Action.Radius(25.0f)
        >] EnemyOrbitingFleetsAction =
           EnemyOrbitingFleetsAction
```

The fight action subtracts 0.5*dt during every tick from the life of the enemy fleets that are within range. All laser targets are added to the LaserTargets list of the source fleet.

```
and [<Action.Action(Target = "Fleet");
        Action.Restrict(Condition = Action.Condition.NotEqual
           , Field = "Owner");
        Action.Transfer(Operation = Action.Operation.Subtract
           , From = "0.5", To = "Life");
        Action.Insert(Value = "this", To = "LaserTargets");
        Action.Radius(150.0f)
        >] FightAction = FightAction
```

The FindNeighbouringFleetsAction adds nearby fleets to the NeighbouringFleets list, so that the flocking algorithm can avoid collisions.

```
and [<Action.Action(Target = "Fleet");
        Action.Insert(Value = "Position", To = "
           NeighbouringFleets");
        Action.Radius(25.0f)
        >] FindNeighbouringFleetsAction =
           FindNeighbouringFleetsAction
```

BuildAction adds GatherSpeed*dt to Minerals inside a planet. When Minerals reaches the 10.0 threshold, then NewFleets is set to true and Minerals is reset to 0.

```
and [<Action.Action(Target = "Self");
       Action.Transfer(Operation = Action.Operation.Add, From
          = "GatherSpeed", To = "Minerals");
       Action.Threshold(Field = "Minerals", Value = "10.0");
       Action.Output(Operation = Action.Operation.Set, Value
          = "true", To = "NewFleet");
       Action.Output(Operation = Action.Operation.Set, Value
          = "0.0", To = "Minerals")>] BuildAction =
          BuildAction
```

Chapter 8

Evaluation

We now present the benchmarks that test the action implementation.

8.0.1 Evaluation

We evaluate the performance of actions with the CS case study of the previous subsection and with an additional example. The additional example is a shooter where the player moves a ship and fires at incoming asteroids. We used actions to model the damage interactions between projectiles and asteroids. Table 8.1 shows a code length comparison between the implementation with actions and standard Casanova rules for CS, Asteroid Shooter and an expanded version of CS with more complex rules, which is used only for comparing the code length and not the performances since the first two samples are enough.

We note that in games with basic dynamics the code saving is not very high due to the fact that the repeated patterns are not many. The advantage of using actions becomes evident in a game with actions involving many types of targets, such as the expanded CS game: without the use of actions, the code related to each interaction would be repeated for each type of target while, with actions, the code is always the same. Furthermore we managed to drastically increase the performance of the game logic: as we can see from Figure 8.1, using R.E.A. (labeled "with actions") gives us a speedup between 6 times and 25 times thanks to automated optimizations in the query evaluation. We also note that our implementation is flexible and general since it is possible to use our actions to express a behavior, such as a projectile collision, which is not strongly related to RTS games, since in those games player do not shoot manually a target but it is the game entity which automatically attacks nearby targets.

Table 8.1: *CS (case study),Asteroid Shooter and Extended CS code length*

	Game Entities	Rules	Actions	Total
CS with REA	41	71	19	131
CS without REA	40	90	0	130
Asteroid shooter with REA	33	33	6	72
Asteroid Shooter without REA	34	44	0	78
Extended CS with REA	135	138	40	313
Extended CS without REA	135	328	0	463

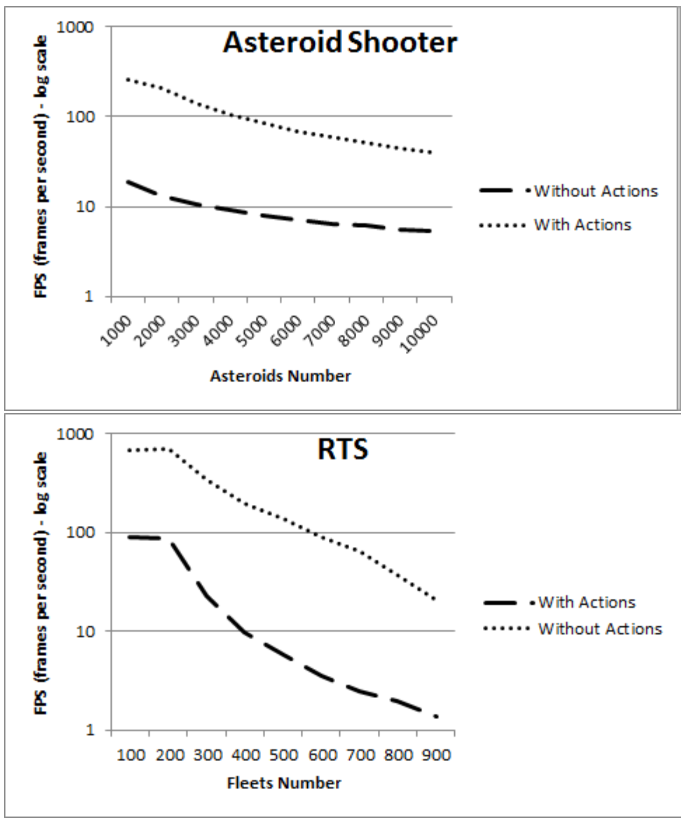

Fig. 8.1: *Frame rate as a function of numbers of entities*

Chapter 9
Conclusions

In this thesis we are concerned with the problem of finding a general way to define RTS games. This problems manifests itself in the boilerplate code that developers have to write when they are rewriting common patterns identified during the generalization process.

The results are:

- A **design pattern** for making RTS games where the interaction among entities can be reduced to a dynamic exchange of resources.
- An expressive, high performance **language extension** to Casanova with a compiler and an appropriate grammar with new syntax and semantic. The leanguage extension is purely declarative. Its semantics resemble SQL.
- An evaluation with three examples provides evidence for increases in programming efficiency.
- The evaluation shows an increase in run time efficiency of 6 to 25 times for the Casanova language, using a native code compiler/opimizer.

Even better results could be obtained with an actual access plan optimizer that would increase the performance when exploring the structure of both the action query and the entity structure. Given the significant results on position indexing, the chance of defining multi-attribute indexes would increase the performance. Finally, a system like F# quotations [38] might be used to increase the expressiveness of the actions.

Bibliography

[1] Adventure construction set. http://en.wikipedia.org/wiki/Adventure_Construction_Set. 26

[2] Arcade game construction kit. http://en.wikipedia.org/wiki/Arcade_Game_Construction_Kit. 26

[3] Brender. http://en.wikipedia.org/wiki/BRender. 26

[4] Entertainment software association. http://www.theesa.com. 1

[5] Garry kitchen's game maker. http://en.wikipedia.org/wiki/Garry_Kitchen's_GameMaker. 26

[6] Pinball construction set. http://en.wikipedia.org/wiki/Pinball_Construction_Set. 26

[7] Rpg maker. http://www.rpgmakerweb.com/. 26

[8] Shoot 'em-up construction kit. http://en.wikipedia.org/wiki/Shoot' Em-Up_Construction_Kit. 26

[9] Shoot 'em-up construction kit. http://en.wikipedia.org/wiki/Shoot' Em-Up_Construction_Kit. 40

[10] Unity reference manual. http://docs.unity3d.com/Documentation/Components/. 40

[11] Unreal engines. http://en.wikipedia.org/wiki/Unreal_Engine. 32

[12] Wargame construction set. http://en.wikipedia.org/wiki/Wargame_Construction_Set. 26

[13] Game engines. http://en.wikipedia.org/, 2012. 4

[14] Clark Aldrich. *The Complete Guide to Simulations and Serious Games: How the Most Valuable Content Will Be Created in the Age Beyond Gutenberg to Google.* Pfeiffer & Co, 2009. 5

[15] Erik Andersen, Yun-En Liu, Rich Snider, Roy Szeto, and Zoran Popović. Placing a value on aesthetics in online casual games. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 1275–1278, New York, NY, USA, 2011. ACM. 5

[16] Khaled Ayad, Dimitrios Rigas, R Rudek, A Rudek, and P Skworcow. Learning with edutainment: A multi-platform approach. In *11th International Conference on Mathematical Methods and Computational Techniques in Electrical Engineering/Applied Computing Conference*, pages 220–225, 2009. 4

[17] T. Barron. Multiplayer game programming. Prima Tech's Game Development. Prima Tech, 2001. 11

[18] Staffan Björk and Jussi Holopainen. *Patterns in game design*. Cengage Learning, 2005. 44

[19] Sue Blackman. *Beginning 3D Game Development with Unity: All-in-one, multi-platform game development*. Apress, 2011. 26

[20] David Blythe. The direct3d 10 system. In *ACM Transactions on Graphics (TOG)*, volume 25, pages 724–734. ACM, 2006. 26

[21] Michael Buro. Real-time strategy gaines: a new ai research challenge. In *Proceedings of the 18th international joint conference on Artificial intelligence*, pages 1534–1535. Morgan Kaufmann Publishers Inc., 2003. 5

[22] Michael Buro and Timothy Furtak. On the development of a free rts game engine. In *GameOn'NA Conference*, pages 23–27. Montreal, 2005. 6

[23] Robert Crandall and J Sidak. Video games: Serious business for america's economy. 2007. 1

[24] R. Barreto de Oliveira. The boo programming language. http://boo.codehaus.org/, 2004. 29

[25] Jason Della Rocca. In the trenches: game developers and the quest for innovation. In *Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames, Boston, MA*, 2006. 4

[26] Jason Gregory. Game engine architecture. Taylor & Francis Ltd., 1 edition, 2009. 11

[27] Juan Gril. The state of indie gaming, 4 2008. 5

[28] Jacob Habgood and Mark Overmars. *The game maker's apprentice: Game development for beginners*. Apress, 2006. 26

[29] Andrew Hunt and David Thomas. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Professional, 2000. 45

[30] Johan Kristiansson. Interview starbreeze studios johan kristiansson. http://www.develop-online.net/features/478/Interview-Starbreeze-Studios-Johan-Kristiansson, 5 2009. 5

[31] David Michael. *Indie Game Development Survival Guide (Charles River Media Game Development)*. Charles River Media, 2003. 5

[32] Jacob Palme. *SIMULA 67: An advanced programming and simulation language*. Oslo: Norwegian Computing Center, 1970. 31

[33] Aaron Reed. *Creating Interactive Fiction with Inform 7*. Course Technology Press, 2010. 32

[34] Aaron Reed. *Learning XNA 4.0: Game Development for the PC, Xbox 360, and Windows Phone 7*. O'reilly media, 2010. 27

[35] John Rice. Assessing higher order thinking in video games. *Journal of Technology and Teacher Education*, 15(1):87–100, 2007. 2

[36] Chris Stoy. Game object component system. *Game Programming Gems*, 6:393–403, 2006. 44

[37] Gregory T Sullivan. Advanced programming language features for executable design patterns" better patterns through reflection. 2002. 44

[38] Don Syme. Leveraging .net meta-programming components from f#: integrated queries and interoperable heterogeneous execution. In *Proceedings of the 2006 workshop on ML*, pages 43–54. ACM, 2006. 79

[39] Günter von Bültzingsloewen. Translating and optimizing sql queries having aggregates. *Proc. 13th Int. Con. VLDB*, pages 235–243, 1987. 59

[40] Walker White, Christoph Koch, Johannes Gehrke, and Alan Demers. Better scripts, better games. *Communications of the ACM*, 52(3):42–47, 2009. 31

[41] Jason Wilson. Indie rocks! mapping independent video game design. Media International Australia Incorporating Culture & Policy, 2005. 4

[42] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL programming guide: the official guide to learning OpenGL, version 1.2*. Addison-Wesley Longman Publishing Co., Inc., 1999. 26