Making Games with Casanova

Giuseppe Maggiore, Alvise Spanò, Renzo Orsini, Michele Bugliesi, Enrico Steffinlongo

Università Ca’ Foscari Venezia

DAIS - Computer Science

{maggiore,spano,orsini,bugliesi,esteffin}@dais.unive.it

**Abstract.** In this paper we present the Casanova language and framework for making games. We show how Casanova is suitable for making games regardless of their genre and we detail how to build a series of simple but very different games: the Game of Life, a shooter game, and a strategy game. We discuss the difference between the Casanova language and the Casanova framework, and we describe the various extensions that we are studying for addition.

**Keywords:** Game development, Casanova, databases, languages, functional programming, F#

1. Introduction

Games are a huge business [1] and a very large aspect of modern popular culture. Independent games, the need for fast prototyping gameplay mechanics [2] and the low budget available for making serious games [3] (when compared with the budget of blockbuster games) has created substantial interest in research on principled design techniques and on cost-effective development technologies for game architectures. Our present endeavor makes a step along these directions.

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 [4]. High quality in games is comprised by two main factors: visual quality and simulation quality. Visual quality in games has made huge leaps forward, and many researchers continuously push the boundaries of real-time rendering towards photorealism; unfortunately, creating a game requires deep knowledge of the arcane inner workings of a GPU and the complex libraries to program one such as DirectX or OpenGL. Simulation quality, on the other hand, is often lacking in modern games; game entities react to the player with little intelligence, and the logic of game levels is often completely linear. Building a high-quality simulation is very complex in terms of development effort and also results in computationally expensive code. To make matters worse, gameplay and many other aspects of the game are modified many times during the course of development while experimenting with the design of the game. For this reason game architectures require a lot of flexibility. To manage all this complexity, game developers use a variety of strategies. Reusable engines mitigate the difficulties of rendering, while object-oriented architectures, external scripting languages, components, reactive programming, etc. have all been used with some degree of success for the crafting of the game logic [5] [6] [7] [8].

In this paper we will present the Casanova language and framework; we will focus more on showing how to build actual games with Casanova than on describing the language itself (which is the focus of other papers [9]). Casanova is the result of a search, started with [9,4], for a general-purpose methodology to make game development easier and speedier, by integrating many of the benefits of the above-mentioned techniques. In the remainder of the paper we show the Casanova language in action. We give a first description of Casanova and we compare it with existing approaches in section [2](#_WHAT_IS_CASANOVA?). We detail much of our framework in section [3](#_THE_CASANOVA_LANGUAGE). We discuss with detailed examples how to make actual games with Casanova in section [4](#_AIMS_OF_CASANOVA). In section [5](#_FUTURE_WORK) we conclude by discussing the main extensions that we are planning on adding to Casanova with our future research.

1. What is Casanova?

The Casanova project actually has two different faces. On one hand is a programming language that offers a mixed declarative and procedural style of programming which has been designed in order to facilitate game development; this language hides most of the low-level details of game development, but it does not (yet) have a fully working compiler implementation. The Casanova language, as of the time of writing, is still just a design tool for *defining and reasoning* about one’s game before diving into actual code, and it can also be considered as a *software engineering* tool. The language allows a designer or an engineer to specify only and exclusively those aspects *specific* to the game being developed, rather than writing lots of boilerplate code. On the other hand, Casanova also offers an actual programming framework; this framework contains a series of managed and native libraries [10] designed to offer many of the language functionalities but in the existing languages of F# and C++ (respectively using the rendering libraries XNA and DirectX 10). Casanova aims for simplicity and expressive power, but it is also designed in order to facilitate or automate optimizations. Casanova ensures consistency in the update of the game world and it also offers integrated support for coroutines, a widespread mechanism in games [11].

Casanova solves the following problems encountered when making a game: *(i)* updating the world *consistently*, that is the world is never stored partially updated; *(ii)* traversing the world fully, that is no portion of the world is left out of an update; *(iii)* expressing queries on the many lists of entities of the world; *(iv)* supporting imperative processes that operate on the game state; *(v)* supporting declarative 2D and 3D rendering.

Each updateable entity of the game state defines the computation that transforms its fields from the current time step to the next. These computations are called rules. Each rule is applied exactly once on each field of each entity it upgrades, and at the same time with all other rules to avoid any temporal interference. This technique allows us to avoid many bugs that derive from the state being updated in place, since it is quite hard to reason about a state that is half in the present and half in the future. Casanova simplifies managing lists of entities by supporting declarative queries (in the style of databases). Queries are optimized with the usual database techniques (with indices, such as spatial partitioning trees). While many aspects of a game lend themselves to a declarative description, other aspects (such as AI, input, level activators, etc.) are more imperative in nature. To support these aspects of a game, Casanova provides support for imperative processes, or *behaviors* (implemented through coroutines) [12]. Finally, some entities may contain instances of predefined drawable data-types such as models, sprites, and texts. Drawable data-types are grouped by their camera when they are 3D models and by layer when they are sprites and text. This grouping specifies common rendering options such as alpha-blending, transforms or even shaders and manages visibility culling. The fields of drawable entities are updated with the same rules that update the other fields of logical game entities.

* 1. Related work

Building a rendering system in a modern game involves, at its core, the building of a scene graph that is fast to traverse for the retrieval of the visual entities of the game. Rendering engines are either built from scratch or licensed from other game studios. Building a new engine is a very hard and time-consuming task that may involve a group of developers several years of constant effort, while using an external rendering engine carries some rather large risks, since engines are large and complex pieces of software that are very hard to use effectively, and which sometimes are difficult to adapt to the evolving needs of the game. In fact, rendering engines tend to be relatively monolithic, that is when an engine is built for a specific type of game then it will be difficult to use for other genres [13].

When it comes to building a game logic, the two most common software architectures are object-oriented hierarchies and component-based systems. In a traditional object-oriented game engine the hierarchy represents the various game objects all derived from the general Entity class. Each entity is responsible for updating itself at each tick of the simulation [6]. A component-based system defines each game entity as a composition of components that provide reusable, specific functionality such as animation, movement, reaction to physics, etc. Component-based systems are being widely adopted, and they are described in [8]. These two more traditional approaches both suffer from a noticeable shortcoming: they focus exclusively on representing single entities and their update operations. By doing so they lose the focus on the fact that most entities in a game need to interact with one another (collision detection, AI, etc.), and usually lots of a game complexity comes from defining (and optimizing) these interactions. Also, all games feature behaviors that take longer than a single tick; these behaviors are hard to express inside the various entities, which often end up storing explicit program counters to resume the current behavior at each tick. Moreover, these architectures simply upgrade everything in place, and offer no guarantees of consistency or against duplicated updates of an entity.

There are two additional approaches that have emerged in the last few years as possible alternatives to object-orientation: (functional) reactive programming and SQL-style declarative programming. Functional reactive programming (FRP, see [7]) is a data-flow approach where value modification is automatically propagated along a dependency graph that represents the computation. While FRP offers a solution to the problem of representing long-running behaviors, it neither addresses the problem of many entities that interact with each other, nor does it address the problem of maintaining the consistency of the game world. SQL-queries for games have been used with a certain success in the SGL language (see [14]). This approach uses a lightweight, statically compiled and heavily optimized query engine for defining a game. SGL suffers when it comes to representing long-running behaviors, since it focuses exclusively on defining the tick function.

We have designed Casanova with these issues in mind: with Casanova, the integration of the interactions between entities and long-running behaviors is seamless, the resulting game world is always consistent, and integrating the visual aspects of the simulation with visibility culling, a scene-graph, and shaders and effects can be done declarative and at very little cost for the developer.

As a final note, this paper builds upon [9], where we presented the original design of the Casanova language. Indeed, the two papers share some similarities. The novel contributions of this paper are: *(i)* a new, simplified syntax for rules that makes them syntactically closer to F# methods; *(ii)* a novel system for managing input with scripted events and responses; *(iii)* a series of in-depth samples and their benchmarks, to show the feasibility of the system in action; and *(iv)* an integrated rendering system for drawing Casanova entities.

1. The Casanova language

The Casanova language is similar to the languages of the ML family (F# in particular, with a few aspects such as list comprehensions inspired from the elegant Haskell syntax) and it is built around the idea of defining a game according to a precise model. List comprehensions have a syntax that is similar to the set notation commonly used in mathematics, and they return the values of an expression evaluated for each combination of values of a series of range variables. Those values from the range variables that fail certain predicates are discarded:

[ expr(x1,x2, …) | x1 <- l1 && x2 <- l2 && … pred1 && pred2 … ]

Casanova is a strongly-typed language, even though it makes use of *type inference* so that type annotations are not always needed (but may still be provided in the form id:type). The type system also supports units of measure, which allow the annotation of certain numeric data-types (floats, vectors, etc.), for example by writing float<s> rather than float<m/s>, and so on.

A Casanova game starts by defining the data-types that describe the game world in a record called World. The game world contains a series of fields that can either be simple variables, other records or collections of values. Each field may have at most one rule attached to it; a rule computes the value that the field will assume *at the next tick* of the game loop. Rules are defined with a pure function that takes as input the state of the world, the entity E and the time between this update and the last one and it returns the new value of the field. Rules are the main work-horses of Casanova games; for example, a rule may update the position of an entity by incrementing it with its velocity:

rule Position(world,e,dt) = e.Position + dt \* e.Velocity

or a rule may cull all the elements of a collection that are dead or disabled:

rule Items(world,e,dt) -> [x | x <- e.Items && e.Disabled = false]

Conceptually, all rules are evaluated simultaneously (and in practice, as well, each rule is assigned to a worker thread that evaluates it in parallel with the other rules). The end result is that all rules are computed on the same game state and they produce the new game state without interferences deriving from their order of evaluation.

After defining the game world data-types and their rules we define the initial state, which is the state of the world when the game is launched and before the first update.

Rules are high-level, expressive constructs and being declarative they allow for many optimizations; this makes them a simple, yet powerful and useful construct. This said, for some aspects of a game a more imperative approach may be needed. To address this shortcoming we have defined a *scripting* system to specify imperative *behaviors* with coroutines, which smoothly integrate with rules. We use a syntax and semantics framework derived from the monadic system of the F# language [15] to build the system of coroutines which define the main imperative process interleaved with the update loop of the game [9]. Coroutines are sequential programs that suspend themselves for the rest of the update cycle whenever they encounter the yield statement. It is worthy of notice that our system is similar to the scripting systems based on coroutines that many games use, even though the degree of integration of our coroutine system with the rest of the game engine is higher when compared with that of commonly used mechanisms which typically attach to the main engine an external scripting language with ungainly binding mechanisms [11]. After defining the initial state of the world, the user provides the main script which represents the main process that will run interleaved with the evaluation of the game rules. Finally, after the main script follow a series of input scripts separated by commas. Each input script is divided into two parts: the event detection and the event response. The event detection is run during each tick of the game; if it returns Some(x) for some x, then x is passed to the event response which is then evaluated; if it returns None, then nothing happens and at the next tick the event detection will be run anew.

Finally, Casanova supports a series of drawable entities which can be stored as a field of a game entity; these drawable entities can be updated with rules or scripts, and are rendered automatically at each tick; entities are grouped into layers (for 2D entities) or cameras (for 3D entities) according to their rendering options, such as shaders, alpha blending, transforms, etc. The camera is also responsible for managing a scene graph and performing visibility culling.

Rendering, rules, consistency, and the behavior system are implemented as a working F# library, and also as a (purely prototypical) C++ library; both may be found at [10]. The F# library has been used extensively in the Galaxy Wars game [16], an upcoming RTS which we have built both as a commercial project and as a research test-bed where the various features of Casanova have been thought, studied, and tested extensively.

It is important to notice that while Casanova may look, on the surface, similar to languages such as C# with LINQ or F#, the support for rendering, scripts, a notion of time and optimized queries are significant differences between Casanova and its (admittedly close) managed cousins when building games.

1. Making games with Casanova

Casanova aims towards making game development easier by integrating patterns that offer various aids to a game developer.

The project, in its current state, is mostly aimed at independent game developers, smaller studios and prototype developers; Casanova might also benefit serious game developers, and in general all those studios that do not have the same development resources typical of major commercial projects. In time the language will offer a standard library of ready-made entities and scripts, it will support game menus, and it will support synchronization of the game state across a network for multiplayer. In short, Casanova aims at taking over all areas of game development, starting from the definition of the game logic and then spiraling outward towards all other areas.

Unfortunately we are aware that a project that aims at making games in general is tackling a very broad problem. For this reason in this paper we present a series of game stubs that implement the core of various very different games, in order to show the flexibility of our system when defining games belonging to different genres. We show how to build: *(i)* the simulation known as *Game of Life*, which we have found to be a good introduction to Casanova; *(ii)* a shooter game where a starship must destroy a series of falling asteroids; *(iii)* an RTS game where the player must conquer various planets by moving his ships from one planet to another.

We will show the various samples in detail, but for space constraints we have greatly simplified the games to the point that their gameplay is quite diminished. The samples (implemented in both Casanova and F# with the Casanova framework for the game logic and using the XNA framework [17] as a rendering back-end) can be downloaded from [10]. Our goal is to show that these Casanova games could be extended into real games, even though with extensive additions; an example of this process can be seen in [16], a strategy game that we are building as an ongoing study of how to create non-trivial games with Casanova.

* 1. Game of Life

The Game of Life, while not properly a videogame (there is no interaction) features many of the aspects of a game: it is a simulation of a virtual world that evolves and changes over time. A matrix of cells is changed according to the following rules: *(i)* a cell with less than two alive neighbors dies because of under-population; *(ii)* a cell with more than three neighbors dies because of over-population; *(iii)* any other cell remains the same.

We start by defining the state of the game as a matrix of cells; the state also contains a boolean variable which will trigger the update of the cell matrix once per second:

type World =

{ Sprites : SpriteLayer

Cells : list<list<Cell>>

UpdateNow : var<bool> }

Each cell contains a value (which is 1 when the cell is alive and 0 when the cell is dead) and a list of its neighbors (marked as ref, since the neighbors of a cell are just references to those cells, rather than the main points of storage for those cells). The value of the cell is updated every time an update is triggered (rather than at each tick), by summing the value of the neighbors and applying the rules mentioned above. The color of the cell is updated to reflect its current value.

type Cell = {

NearCells : list<ref<Cell>>

Value : int

Sprite : DrawableSprite {

rule Value(world,self,dt) =

if state.UpdateNow then

let around = sum [c.Value | c <- self.NearCells]

match around with

| 3 -> 1

| 2 -> self.Value

| \_ -> 0

else

self.Value

rule Sprite.Color(world,self,dt) = if self.Value = 0 then Color.Black else Color.White

The initial state of the game creates the matrix of cells and initializes their neighbors. Each internal cell has exactly eight neighbors. The sprites layer and the cell sprite are also setup for each sprite. Notice that when assigning the fields of a rendering entity or layer, we just specify those parameters we are interested in; all other parameters are assigned default values.

let initial\_world =

let make\_cell i j =

{ NearCells = []

Value = random(0, 4) / 3

Sprite = { Path = "WhitePixel.bmp"

Position = vector2(i,j)

Layer = sprites }

let sprites = { Transform = Matrix.Identity; AlphaBlend = false }

let world\_aux =

{ Sprites = sprites

Cells =

[ for i = 1 to 100 do yield [ for j = 1 to 100 do yield make\_cell i j } } ] ]

UpdateNow = false }

… // setup neighbors

world\_aux

The rules of the game are fired at every frame of the game that is roughly 60 times per second. Changing the entire matrix of cells this often would yield a chaotic result; for this reason we have defined the UpdateNow field in the game state, so that we can control when the rules are fired.

The main script of the game waits for a second before toggling the UpdateNow value, and then it suspends itself until the next iteration of the update loop. When the script is resumed, it toggles UpdateNow again and finally it repeats:

let main world =

repeat {

wait 1.0

world.UpdateNow := true

yield

world.UpdateNow := false }

This way the game of life will run at exactly one step per second, no matter the framerate of the simulation.

* 1. Asteroid Shooter

The asteroid shooter game is a simple shooter game where asteroids fall from the top of the screen towards the bottom. The player aims the cannon and shoots the asteroids to prevent them from reaching the bottom of the screen.

The logical fields of the game world are: *(i)* a list of asteroids, which rule produces the collection of updated asteroids except those that reach the bottom of the screen or that hit a projectile (which are thusly removed); *(ii)* a list of projectiles, which are removed when they reach the top of the screen or when they hit an asteroid; *(iii)* the current direction the cannon is aiming at, which is updated via the user input and *(iv)* the score counter. The game world also contains a series of rendering fields: *(i)* the background layer and a stars texture; *(ii)* the 3D camera to draw the asteroids, projectiles and cannon models; *(iii)* the ui layer and the current score.

type World = {

Background : SpriteLayer

ModelsCamera : Camera

UI : SpriteLayer

StarsSprite : DrawableSprite

ScoreText : DrawableText

Asteroids : var<list<Asteroid>>

Projectiles : var<list<Projectile>>

Cannon : Cannon

Score : int }

rule Asteroids(world,dt) =

[a | a <- state.Asteroids && a.Colliders.Length = 0 && a.Position.Y < 100.0<m>]

rule Projectiles(world,dt) =

[p | p <- state.Projectiles && p.Colliders.Length = 0 && p.Position.Y > 0.0<m>]

rule Score(world,dt) =

world.Score + [a | a <- state.Asteroids && a.Colliders.Length > 0].Length

rule ScoreText.String(world,dt) = "Current score = " + world.Score.ToString()

The cannon rotates left and right according to user input:

type Cannon = {

Model : DrawableModel

Angle : float<rad>

TurnLeft : var<bool>

TurnRight : var<bool> }

rule Angle(world,self,dt) =

self.Angle + if self.TurnLeft then dt elif self.TurnRight then -dt else 0.0<rad>

rule TurnLeft(world,self,dt) = false

rule TurnRight(world,self,dt) = false

rule Model.Rotation(world,self,dt) =

Quaternion.FromYawPitchRoll(self.Angle,0.0<rad>,0.0<rad>)

Asteroids and projectiles are very similar, so instead of giving the same definition twice we define a physical entity which moves according to its velocity and collides with other entities in the game world:

contract PhysicalEntity<'c:PhysicalEntity,get\_colliders:World->list<'c>> = {

Model : DrawableModel

Position : vector2<m>

Velocity : vector2<m/s>

Colliders : list<ref<'c>> }

rule Position(world,self,dt) = self.Position + self.Velocity \* dt

rule Colliders(world,self,dt) =

[x | x <- get\_colliders world && distance(self.Position, x.Position) < 10.0f]

rule Model.Position(world,self,dt) = vector3(self.Position.X, self.Position.Y, 0.0<m>)

type Asteroid = PhysicalEntity<Projectile, fun world -> world.Projectiles>

type Projectile = PhysicalEntity<Asteroid, fun world -> world.Asteroids>

The initial world state contains no asteroids and no projectiles, the cannon points upwards and the score counters are both zeroed; also, the various rendering and the camera layers are initialized. We do not show the source for the initial state, as it is trivial.

The main script generates random asteroids in a loop:

let main world =

repeat {

wait (random(1.0<s>,3.0<s>))

state.Asteroids.Add {

Model = { Path = "asteroid.fbx"

Camera = world.ModelsCamera }

Position = vector2(random(0.0<m>,100.0<m>),0.0<m>)

Velocity = vector2(0.0<m/s>,random(5.0<m/s>,20.0<m/s>))

Colliders = [] } }

Projectiles are generated by waiting for the user to press the space button; when space is pressed, then a projectile is created with a velocity that corresponds to the current aim of the cannon. To avoid generating one projectile every frame, we wait one-fifth of a second every time a projectile is generated; this way even if the user holds the space button for a long period of time the number of projectiles shot every second will be constant (namely, 5). Other input scripts handle aiming the cannon:

let input world =

{ if is\_key\_down Keys.Space then return Some() else return None } => {

state.Projectiles.Add

{ Model = { Path = "projectile.fbx"

Camera = world.ModelsCamera }

Position = vector(50.0<m>, 0.0<m>)

Velocity = vector2(cos(state.CannonAngle),sin(state.CannonAngle))

Colliders = [] }

wait 0.2<s> },

{ if is\_key\_down Keys.Left then return Some(true) else return None } => {

state.Cannon.TurnLeft := true },

{ if is\_key\_down Keys.Right then return Some(true) else return None } => {

state.Cannon.TurnRight := true }

* 1. Strategy Game

The strategy game features a series of planets that produce ships, which can then be sent to conquer other planets. The following presentation is actually the heavily reduced rewriting of the much larger (tens of thousands of lines of code) upcoming strategy game Galaxy Wars [16] that we have developed both as a commercial endeavor and as a research test-bed for Casanova; the game features thousands of simultaneous entities, real-time 3D graphics with special effects, and even real-time multiplayer via LAN and Internet of up to 8 players. The game clearly shows the feasibility of Casanova when used on a larger scale.

The state of the game contains a list of planets and a list of fleets. Fleets are culled from the state when they reach their destination and finish fighting. Similarly to the Game of Life, we use a variable to inhibit the application of some rules (those that implement the battles) so that they do not change the state too quickly. The world also contains two sprite layers: one for planets and fleets, another for the numbers that indicate the strength of each unit. The world also contains the currently selected planet as indicated by the user with a mouse click.

type World = {

Sprites : SpriteLayer

UI : SpriteLayer

Planets : list<Planet>

Fleets : var<list<Fleet>>

TickBattles : var<bool>

SourcePlanet : var<Option<ref<Planet>>> }

rule Fleets(world,dt) =

[f | f <- self.Fleets && f.Alive && (not(f.Arrived) || f.Fighting)]

A planet contains an owner, its position, the current number of armies it hosts, the progress in building the next army, the production rate and the list of enemy fleets inbound to the planet. The owner of the planet and the current armies are modified according to the result of battle against the attacking fleets; when the planet armies are all destroyed, then the planet is owned by the owner of the first attacking fleet. Friendly fleets add armies to the planet upon their arrival. Notice how the number of armies is an integer with the custom Ship unit of measure.

type Planet = {

Position : vector2<km>

Owner : Player

Armies : var<int<Ship>>

FractionalArmies : float<Ship>

AttackingFleets : list<ref<Fleet>>

ReinforcingFleets : list<ref<Fleet>>

Production : float<Ship/s>

Sprite : DrawableSprite

ArmiesText : DrawableText }

rule Owner(world,self,dt) =

if self.Armies <= 0 && self.AttackingFleets <> [] then self.AttackingFleets[0].Owner

else self.Owner

rule Armies(world,self,dt) =

if self.Armies <= 0 then

sum [a.Armies | a <- self.AttackingFleets && a.Owner = self.AtackingFleets[0].Owner]

else

let damages = if state.TickBattles then sum[random(1,3) | f <- self.AttackingFleets]

else 0.0<Fleet/s>

let reinforcements = sum[f.Armies | f <- self.ReinforcingFleets]

self.Armies + int(self.FractionalArmies) - damages + reinforcements

rule FractionalArmies(world,self,dt) =

self.FractionalArmies + (dt \* self.Production) - floor(self.FractionalArmies)

rule AttackingFleets(world,self,dt) =

[f : f <- state.Fleets && f.Target = self && f.Owner <> self.Owner && f.Arrived]

rule ReinforcingFleets(world,self,dt) =

[f : f <- state.Fleets && f.Target = self && f.Owner = self.Owner && f.Arrived]

rule ArmiesText.String(world,self,dt) = self.Armies.ToString()

Fleets contain a position, an owner, a velocity, a target, a number of armies and various indicators that determine the current state of the ship: whether it is fighting, traveling and still alive. The ship updates its position according to its velocity and its armies according to battle:

type Fleet = {

Position : vector2<km>

Velocity : vector2<km/s>

Owner : Player

Target : ref<Planet>

Arrived : bool

Armies : int<Ship>

Alive : bool

Sprite : DrawableSprite

ArmiesText : DrawableText }

rule Position(world,self,dt) = self.Position + self.Velocity \* dt

rule Velocity(world,self,dt) = if self.Arrived then vector2.Zero else self.Velocity

rule Arrived(world,self,dt) = distance(self.Position, self.Target.Position) < 100.0<km>

rule Armies(world,self,dt) =

if self.Arrived && self.Target.Owner <> self.Owner &&

self.Alive && state.TickBattles then self.Armies – random(1,3)

else self.Armies

rule Alive(world,self,dt) = self.Armies > 0<Ship>

rule Sprite.Position(world,self,dt) = self.Position

rule ArmiesText.String(world,self,dt) = self.Armies.ToString()

The game only features two players: a human and an AI:

type Owner = Human | AI

The initial state of the game creates a set of planets that can either be preset or loaded from a file; at the beginning of the game there is no fleet active.

The main script of the game is responsible for creating the various fleets according to user input and a rudimentary AI:

let mk\_fleet world source target =

... // send half the armies of source against target in a new ship

// assign ship sprite to sprites\_layer and text sprite to ui\_layer

let main world =

repeat {

wait 1.0

world.TickBattles := true

yield

world.TickBattles := false

let own\_planets = [p | p <- state.Planets && p.Owner = AI]

let enemy\_planets = [p | p <- state.Planets && p.Owner<>AI]

if enemy\_planets <> [] && own\_planet <> [] then

let source = List.maxBy (fun p -> p.Armies) own\_planets

let dest = List.minBy (fun p -> p.Armies) enemy\_planets

if source.Armies > 0 then mk\_fleet world source dest } }

The input script waits for the user to left click on a planet (the source) and then right click on another (the target) to send armies from one to the other if the source owner is the human player.

let input world =

{ if mouse\_clicked\_left() then

let mouse = mouse\_position()

let clicked = [p | p <- state.Planets && distance(p.Position,mouse) < 10.0]

if clicked <> [] then return Some(clicked.[0])

else return None } => fun p -> { world.SourcePlanet := Some(p) },

{ if mouse\_clicked\_right() && world.SourcePlanet <> None then

let mouse = mouse\_position()

let clicked = [p | p <- state.Planets && distance(p.Position,mouse) < 10.0]

if clicked <> [] then return Some(clicked.[0],world.SourcePlanet.Value)

else return None } => fun (source,target) -> { mk\_fleet world source target }

* 1. Consistency and automated optimizations

The asteroids game in particular shows two very important features of Casanova: consistency and automated optimizations. As a simple example, consistency comes into play when we are computing collisions between projectiles and asteroids. If we removed asteroids from the game state whenever we found those asteroids to be in collision with projectiles, then those asteroids would not be found anymore when determining which projectiles to remove. Let us consider this example:

asteroids = [a1; a2; a3]  
projectiles = [p1; p2]

We check the asteroids for collision, and we find that a2 collides with p1; we update the state so that:

asteroids = [a1; a3]  
projectiles = [p1; p2]

Unfortunately now we do not find the collision between p2 and a2 anymore because a2 has been removed from the set of current asteroids. While this specific bug is not too hard to fix once identified, it belongs to a larger class of common bugs that happen whenever the state of the game is only partially updated, and is representing part of the world at time step t and part at time step t+1. If the state of the game is large and the order in which the various game entities are updated is not fixed (for example if sometimes asteroids are updated before projectiles, and other times the opposite), then this can become a major source of unpredictable errors that are difficult to reproduce, diagnose and fix. The second, extremely important feature of Casanova is that of automated optimization of quadratic queries. Rules of the form:

rule Field(state, self, dt) -> [ x | x <- state.Xs && cond x self ]

are optimized by building an index (be it a spatial partitioning index such as a quad-tree or a hash-map) that makes determining the Xs that satisfy the predicate cond much faster. Building this index can give dramatic performance increases. For the asteroids game, using this optimization gives a huge performance boost in a stress test: the framerate without the optimization is 8 fps, while the framerate with the optimization jumps to 577 fps (which is actually more than we expected). In similar stress tests, the RTS goes from 27 fps to 202 fps. Moreover, since no rule writes the same memory location of another, rules can be computed in parallel, producing another noticeable speedup in the order of up to the number of available cores for certain programs. This is actually very simple to build: the various rules to compute are assigned to a series of worker threads, which then evaluate them in parallel. The last optimization is visibility culling performed by the camera to avoid rendering those 3D models that are not visible on the screen. Also, by grouping drawable entities in layers, we can save some computational power by avoiding many operations that set rendering options on the graphics card, as this amounts to a form of *batching* [18].

1. Future Work

We believe our work to have opened exciting new venues of exploration. Casanova started with the goal of making it simpler to build a declarative, easily optimized game logic, with its associated rendering. In addition to completing support for the Casanova language, both as a standalone compiler and as a code generator built with the upcoming *type providers* F# feature, we will also try to tackle further tasks: *(i)* we are studying a *Casanova Standard Library* of ready-made reusable entities to make the creation of a game even simpler; *(ii)* we are studying how to add networking to Casanova for multiplayer games; *(iii)* we are studying the integration of AI techniques such as [19]; *(iv)* we have a long list of query optimizations [20] that could make Casanova more efficient; *(vi)* we plan on adding a visual editor for scripts so that non-technical users may create scripts without having to learn a new programming language. Furthermore, we are planning to assess the ease of use of Casanova when compared with traditional frameworks such as XNA or Unity. We have already moved a few preliminary steps in this direction, both with High School students and first year university students: Casanova programs are shorter (by one third on average when compared with F#), faster and easier to write than F# and XNA.

1. Conclusions

Game development is a large and important aspect of modern culture; games are used for entertainment, education, training and more, and their impact on society is very large. This is driving a need for structured principles and practices for developing games and simulations. Also, reducing the cost and difficulties of making games could greatly benefit some “fringe” game developers, such as independent game developers, serious game developers, and even research game developers, who traditionally have neither the budget nor the manpower to tackle some of the challenges associated with making a modern game. Casanova is a step in this direction: by studying the art and craft of game development we are building a framework and a language that simplify many tasks and allow game developers to put more effort on AI, gameplay, shaders and other important tasks such as procedural generation rather than on the “nuts-and-bolts” of putting a game together and optimizing it. While Casanova is still in its early stages, we have used it extensively and with good results in a real game [16], and we are certain that with further work the benefits of this approach will become much more apparent.

1. References

x

|  |  |
| --- | --- |
| 1. | ENTERTAINMENT SOFTWARE ASSOCIATION. **Industry Facts**. [S.l.]: [s.n.]. 2010. |
| 2. | FULLERTON, T.; SWAIN, C.; HOFFMAN, S. **Game design workshop:** a playcentric approach to creating innovative games. [S.l.]: Morgan Kaufman, 2008. ISBN 0240809742. |
| 3. | RITTERFELD, U.; CODY, M.; VORDERER, P. **Serious Games:** Mechanisms And Effects. [S.l.]: Routledge. 2009. |
| 4. | BUCKLAND, M. **Programming Game AI by Example**. Sudbury, MA: Jones & Bartlett Publishers. 2004. |
| 5. | WILSON, K. **Inheritance vs aggregation in game objects**. http://gamearchitect.net/Articles/GameObjects1.html. [S.l.]: [s.n.]. 2002. |
| 6. | AMPATZOGLOU, A.; CHATZIGEORGIOU, A. **Evaluation of object-oriented design patterns in game development**. Journal of Information and Software Technology. MA, USA: Butterworth-Heinemann Newton. 2007. |
| 7. | CONAL, E.; HUDAK, P. **Functional reactive animation**. International Conference on Functional Programming (ICFP). [S.l.]: [s.n.]. 1997. p. 263–273. |
| 8. | FOLMER, E. **Component based game development:** a solution to escalating costs and expanding deadlines? Proceedings of the 10th international conference on Component-based software engineering, CBSE. Berlin, Heidelberg: Springer-Verlag. 2007. p. 66– 73. |
| 9. | MAGGIORE, G. et al. **Designing Casanova:** a language for games. In Proceedings of the 13th conference on Advances in Computer Games, ACG 13, Tilburg, 2011, Springer. 13th Internation Conference Advances in Computer Games (ACG). Tilburg, Netherlands: Springer. 2011. |
| 10. | MAGGIORE, G. **Casanova project page**. http://casanova.codeplex.com/. [S.l.]: [s.n.]. 2011. |
| 11. | FIGUEIREDO, L. H. D.; CELES, W.; IERUSALIMSCHY, R. **Programming advance control mechanisms with Lua coroutines**. Game Programming Gems 6. [S.l.]: Mike Dickheiser (ed), Charles River Media. 2006. p. 357–369. |
| 12. | KNUTH, D. E. **The art of computer programming**. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc. 1997. |
| 13. | DELOURA, M. The Engine Survey. **Gamasutra**, 2009. Disponivel em: <http://www.gamasutra.com/blogs/MarkDeLoura/20090316/903/The\_Engine\_Survey\_Technology\_Results.php>. |
| 14. | WALKER WHITE, A. D. C. K. J. G. A. R. R. **Scaling games to epic proportions**. Proceedings of the 2007 ACM SIGMOD international conference on Management of data (SIGMOD). New York, NY, USA: ACM. 2007. p. 31–42. |
| 15. | COSTANTINI, G.; MAGGIORE, G. **Friendly F# (fun with game programming)**. Venice, Italy: Smashwords. 2011. |
| 16. | MAGGIORE, G. **Galaxy Wars Project Page**. http://vsteam2010.codeplex.com. [S.l.]: [s.n.]. 2010. |
| 17. | CORP., M. **The xna framework**. 2004: http://msdn.microsoft.com/xna. |
| 18. | PHARR, M.; FERNANDO, R. **GPU Gems 2 - Chapter 2:** Inside Geometry Instancing. [S.l.]: Addison-Wesley Professional, 2005. |
| 19. | RICHARD ZHAO, D. S. **Generating Believable Virtual Characters Using Behaviour Capture and Hidden Markov Models**. 13th Internation Conference Advances in Computer Games (ACG). Tilburg, Netherlands: Springer. 2011. |
| 20. | GARCIA-MOLINA, H.; ULLMAN, J. D.; WIDOM, J. **Database System Implementation**. [S.l.]: Prentice-Hall. 1999. |

x