On Casanova and Databases or the Similarity Between Games and DBs*

Giuseppe Maggiore, Renzo Orsini, and Michele Bugliesi

Università Ca' Foscari Venezia
DAIS – Computer Science
{maggiore,orsini,bugliesi}@dais.unive.it

Abstract. In this paper we discuss the similarities between two fields which are traditionally considered worlds apart: game development and databases. We discuss how many aspects of game development either use databases, data-mining, etc. directly to solve challenging data-management problems, but also how the game logic itself is subtly related to many techniques and theoretical results already explored in the field of databases. We also discuss our Casanova language, which is a game development language which we are building with the explicit aim of taking advantage of this relationship, in order to greatly simplify the craft of game-making by introducing automated optimizations and declarative constructs to define a game with less boilerplate code.

Keywords: Game development, Casanova, databases, languages, functional programming

1 Introduction

Games are a growing field, rapidly approaching in size and audience the music and movie industries [1]. Game development goes beyond entertainment: serious games [2] experiment with the use of gameplay to teach important lessons, while [3] even use interactive game development to teach Computer Science to young students. Also, the same tools and techniques used to create games are used for virtual reality and interactive simulations in general. Unfortunately games are very much unexplored territory when it comes to disciplined research, a research that would be much needed to contain the skyrocketing costs involved with creating a modern game. Reducing the costs of game-making would allow researchers and practitioners alike to explore new and interesting games without the risk of a very large initial investment. This may indeed be the reason why a large number of modern commercial games destined to the mass-market are built with so little daring in terms of exploring new gameplay and new mechanics: to reduce the risks of exploring new territories.

Where does the difficulty in making games come from? Game development is so expensive because to ensure that the final result stands up to the user expectations [4] a game needs a high visual quality to clearly show the various states of the logical entities of the game, and the logical entities of the game must be updated according to

* Extended Abstract

Nicola Ferro and Letizia Tanca (Eds.): SEBD 2012, Edizioni Libreria Progetto, Padova, Italia
ISBN: 978-88-96477-23-6,

an articulated simulation that evolves their state in a meaningful way. The visual and logical modules of the game are large and complex to build; to make matters worse, both must run in a loop that is optimized enough to update the screen and animate the game entities in real-time, that is all iterations of the game logic and drawing must be completed in a time span that ranges between $1/20^{th}$ and $1/60^{th}$ of a second. As an additional challenge, game-making comprises a (rather large) creative portion that is performed by designers, who rarely are well-versed in the arcana of computer programming: for this reason the architecture of a game must be flexible and easily modifiable so that designers can quickly build and test new iterations of gameplay. Finally, the surrounding system of a modern multiplayer game poses additional challenges, such as synchronizing the game world across many clients, *always in real-time*, clustering players by skill-level to create balanced matches, and so on. In short, games offer a unique blend of complexity, optimization, and need for customization by non-programmers which yields very high construction and maintenance costs.

In this paper we present a novel observation: many game development problems may be already solved in a field which, at a first glance, may appear utterly unrelated. The field of databases already contains a large body of relevant research works which simply needs to be studied and adopted by game developers. For this purpose we are creating a game development framework, called Casanova [5,6,7], that simplifies game development along this direction. We will start with a discussion about coding the logical simulation of a game in Section 2; we then study the matters of world persistency in a "massively multiplayer online game" (MMOG) in Section 3; finally, we conclude with a remark on mining players data for match-making or preference deduction in Section 4.

1.1 Related work

There is at least one other underway research effort of linking database research with game development; this work has yielded the SGL language [8], an experimental game development language which uses SQL queries to define the way the various entities of the game world are updated at each time-step of the simulation. SGL may be unsuitable for larger scale problems, since it offers no techniques to model the game world and entities, but the underlying optimizations and expressivity of the framework are undeniably very powerful and require virtually no effort on the part of the game developer.

The present work also builds on our database-inspired game development language, Casanova [5] [7] [6]. The language aims at offering a series of abstractions and optimizations that allow a developer to specify only certain core aspects of a game logic and visualization, without concerning himself too much with boilerplate code such as state traversal or query optimizations.

Nicola Ferro and Letizia Tanca (Eds.): SEBD 2012, Edizioni Libreria Progetto, Padova, Italia
ISBN: 978-88-96477-23-6,

2 The Game World

The logical simulation of a game starts exactly at the first step of the creation of a new database: modeling (or conceptual schema definition [9]). A game consists, at its core, of a series of concepts and their relationships. This describes the semantics of a game world and represents a series of assertions about its nature. Specifically, it describes the things of significance to the game, about which it is inclined to collect information, and characteristics of (*attributes*) and associations between pairs of those entities (*relationships*). Most entities are in the plural, and thus require being stored in tables or collections. For example, the Game of Life might be modeled in Casanova as:

```
type World = { Cells : list<list<Cell>> }
type Cell = {
  NearCells   : list<ref<Cell>>
  Value       : int }
```

After defining the data model of the game world, game developers must define the *dynamics* of the game, that is how each game entity is updated at every tick of the game loop. The game dynamics is, at its core, a series of rules that define how each entity (or, better, each attribute of each entity) is updated during each tick. A major point of difference between games and databases lies in the frequency of the dynamics of the system: the game world is updated about once every sixtieth of a second to achieve a smooth simulation, instead of waiting for user-initiated events; indeed, a large number of changes in the game world are entirely automated and occur naturally over time. In Casanova the game dynamics is computed by stating a series of rules for each field of the game that needs updating, that is the above definition would also include:

```
type Cell = {
  NearCells   : list<ref<Cell>>
  Value       : int }
 rule Value(world,self,dt) =
  let around = sum [c.Value | c <- self.NearCells]
  match around with
  | 3 -> 1
  | 2 -> self.Value
  | _ -> 0
```

Some of the game dynamics simply require to recomputed simple values, while others require more sophistication. Consider a game where we need to compute the collisions between asteroids and projectiles; we might define a projectile so that it computes a query on the entire game world to find the list of asteroids colliding with itself at each tick:

```
type Projectile = {
 Position       : vector2<m>
 Velocity       : vector2<m/s>
 Colliders      : list<ref<Asteroid>> }
 rule Position(world,self,dt) = self.Position + self.Velocity * dt
 rule Colliders(world,self,dt) =
  [x | x <- world.Asteroids && distance(self.Position, x.Position) < 10.0f]
```

Notice that certain attributes of the game entities are marked with the ref data constructor, which represents referential constraints (foreign keys) [10] between different lists of entities; references define attributes which do not contain entities to be updated during a tick; in our example above this means that the colliders of a projectile are

Nicola Ferro and Letizia Tanca (Eds.): SEBD 2012, Edizioni Libreria Progetto, Padova, Italia
ISBN: 978-88-96477-23-6,

not asteroids to be updated during a tick, but just a series of asteroids which we need for certain logical computations to be associated with a certain projectile.

Rules are treated as transactional operations [11] in order to ensure the consistency of the game world. This means that all rules are evaluated on the game world at a certain time-step ($world_t$) and then all their results are written, at the same time, into the new game world ($world_{t+1}$). This way all rules behave in a predictable way and no rule ever "sees" the game world halfway between different ticks of the simulation. Moreover, this enables a very important optimization: evaluating rules in parallel with different threads so as to speed up the simulation, thus freeing computational power to animate more entities or use more complex algorithms.

Rules on collections also present an additional optimization opportunity: certain operations (the colliders example above is a particularly fitting example) need to compute a Cartesian product between two lists, asteroids and projectiles, which naïvely computed would have quadratic complexity. By using optimization techniques such as a hash-join or similar the complexity becomes much lower. Our benchmarks [7] suggest improvements of an order of magnitude in the run-time efficiency of the entire simulation when applying query optimization techniques.

Rules and queries are not always the best abstraction to represent the way a game world evolves itself over time. For this reason we have added to Casanova a scripting system, which is decidedly akin to a system of triggers and stored procedures (where triggers may also be timers or user actions). The (soft) real-time constraint of a game requires that our procedures do not block a game tick for an excessive period of time, because otherwise this would break the smoothness of the user experience. To better mix the game loop and the evaluation of these procedures we have implemented them as coroutines [5], that is they feature a yield statement that suspends procedure evaluation until the next tick, and the wait statement that suspends procedure evaluation for a given amount of time. We would define a Casanova script that waits for the player to press a button to shoot a projectile by writing:

```
{ if is_key_down Keys.Space then return Some() else return None } => {
  world.Projectiles.Add
    { Position = vector(50.0<, 0.)
      Velocity = vector2(cos(state.CannonAngle),sin(state.CannonAngle))
      Colliders = [] }
  wait 0.1<s> }
```

3 Persistency, Saving Games and Multiplayer Games

A game world requires some persistency. Persistency comes into play both in single-player games and multi-player games. Single-player games require persistency because the playing experience takes much longer than a single play session, and so the game state requires serialization on persistent memory. The act of storing and retrieving the game world from persistent memory must be quick, since saving the game can be (and often is) done during gameplay and thus must be optimized for speed as the rest of the game, to avoid breaking the flow of gameplay.

A more complex case where the game world is persistent is that of multiplayer games. Multiplayer games have two different sets of problems to tackle: *(i)* synchronizing the

Nicola Ferro and Letizia Tanca (Eds.): SEBD 2012, Edizioni Libreria Progetto, Padova, Italia
ISBN: 978-88-96477-23-6, Copyright (c) 2012 - Edizioni Libreria Progetto and the authors

game world in real-time between different clients; and *(ii)* reliably storing a persistent world and all the players' data.

Synchronization of the game world between many clients and the game server (or *host*) must happen in real-time, but each client needs a responsive experience. For this reason most modern games employ client-side prediction and lag-compensation algorithms [12], that is all operations that need to write the host' game world always appear to succeed locally and is then validated (as soon as possible considering the roundtrip time for unreliable networked messages) by the host. This amounts to a form of eventual consistency.

The problem of storing a persistent, huge world for many players (games such as World of Warcraft feature *millions* of concurrent players) requires hybrid in-memory/on-disk databases with very quick access and supporting up to hundreds of thousands of concurrent accesses. To reduce the scope of these technical challenges the game world is sometimes segmented into different copies of the world, grouping players by geographical reason, but other games such as EVE Online feature different techniques such as a hierarchical structure of distributed servers to avoid segmentation and offer a single persistent game world.

4 Match-making, Understanding Players

Another challenge that multiplayer games face is that of making use of the huge amount of data that can be gathered from players' behavior through data-mining techniques. A common instance of this is the match-making problem [13]: given the preferences of the players who are currently waiting to start a game, determine the ideal group of players who all have similar skills, low-latency between each other, similar preferences, etc. Some games even feature team-play, and so the best teams must be defined automatically.

Similarly, game developers often need to understand the preferences of their players or if there are certain conditions in the game that favor certain players, in order to maintain a fun, balanced and fair experience for everyone. Discriminating useful patterns from previous games logs requires the ability to wade through huge data bases to make sense of their information.

5 Conclusions and Future Work

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.

Nicola Ferro and Letizia Tanca (Eds.): SEBD 2012, Edizioni Libreria Progetto, Padova, Italia
ISBN: 978-88-96477-23-6,

Modern games often intersect with databases, both when constructing the core of the simulation and managing the massive amounts of (often distributed) information associated with a game. By building awareness of this relationship we hope to encourage more database researchers to help share the "wisdom of their trade" with game developers, creating a fruitful exchange of knowledge and offering new viewpoints to older problems.

References

1. Entertainment Software Association: Industry Facts. (2010)
2. Ritterfeld, U., Cody, M., Vorderer, P.: Serious Games: Mechanisms And Effects. (2009)
3. Conway, M., Pausch, Y., Gossweiler, R., Burnette, T.: Alice: A Rapid Prototyping System for Building Virtual Environments. (1995)
4. Buckland, M.: Programming Game AI by Example., Sudbury, MA (2004)
5. Giuseppe Maggiore, M.: Monadic Scripting in F# for Computer Games., Oslo, Norway (2011)
6. Maggiore, G., Spanò, A., Orsini, R., Costantini, G., Bugliesi, M., Abbadi, M.: Designing Casanova: a language for games. In Proceedings of the 13th conference on Advances in Computer Games, ACG 13, Tilburg, 2011, Springer. In : 13th Internation Conference Advances in Computer Games (ACG), Tilburg, Netherlands (2011)
7. Maggiore, G., Bugliesi, M., Orsini, R.: Casanova Papers. In: Casanova project page. (Accessed 2011) Available at: http://casanova.codeplex.com/wikipage?title=Papers
8. Walker White, A.: Scaling games to epic proportions. In : Proceedings of the 2007 ACM SIGMOD international conference on Management of data (SIGMOD), New York, NY, USA, p.31–42 (2007)
9. Perez, S., Sarris, A.: Technical Report for IRDS Conceptual Schema, Part 1: Conceptual Schema for IRDS, Part 2: Modeling Language Analysis. (1995)
10. Garcia-molina, H., Ullman, J., Widom, J.: Database System Implementation. (1999)
11. Weikum, G., Vossen, G.: Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery. (2001)
12. Bernier, Y.: Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization. https://developer.valvesoftware.com/wiki/Latency_Compensating_Methods_in_Client/Server_In-game_Protocol_Design_and_Optimization (2001)
13. Trelford, P.: Learning with F#. In : Proceedings of the 4th ACM SIGPLAN workshop on Commercial users of functional programming (2007)

Nicola Ferro and Letizia Tanca (Eds.): SEBD 2012, Edizioni Libreria Progetto, Padova, Italia
ISBN: 978-88-96477-23-6,