Query languages applied to game development

Giuseppe Maggiore
Giulia Costantini
Università Ca' Foscari Venezia
Dipartimento di Scienze Ambientali,
Informatica e Statistica
{maggiore,costantini}@dais.unive.it

May 20, 2011

Abstract

In this report we summarize the application of declarative query languages to game development. Thanks to the use of a language that is very similar to SQL the authors have been able to build games with complex logic, fast runtime and very little and easily readable code.

1 Introduction

Computer games are growing in importance at an astonishing speed. Games have since long catched up with Hollywood in economic terms [1], their use base for entertainment purposes is huge [19] and even more importantly games are more and more being studied as the next frontier in engaging education [9, 12].

Up until not many years ago, computer graphics were the main source of innovation in games. As realtime computer graphics is approaching an exceptionally high mark some focus is shifting away from making better looking games and is instead focusing on *the interactive experience*. An example of this phenomenon is the fact that the current generation of consoles has moved away from the traditional 4-year update cycle and instead has experimented with novel forms of user interaction [10].

The papers we discuss [22, 5, 21, 20] support this view, and add that a very important aspect of the interactive experience is the AI (artificial intelligence) of the animated characters that play the part of the various actors of the game. Traditionally, games have improved game AI along two axes:

- character depth;
- number of characters.

Character depth has been used in order to build fewer characters that interact with the user for a long duration. These characters must be believable, and

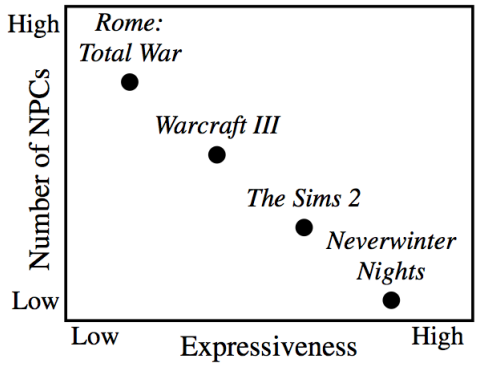

Figure 1: AI complexity versus number of characters

so they must behave smartly and sometimes even convey strong emotions: the story villain and the player's small party of followers must all be engaging and credible. To this end, expert systems have been employed to various degrees in games [11, 6, 13].

Deep characters are very computationally expensive. Modern AI techniques would be unfeasible if applied to more than a handful of such characters. Strategy games and massively multiplayer online role playing games (and open world games in general) require AI techniques that are capable of supporting hundreds or thousands of animated characters [4, 16]. In these games character behaviors are often governed by simple state machines. The combination of many such simple AIs makes for interesting emergent behaviors that combine into a compelling gameplay. A risk that is ever-present is that of dumbing down the AI intelligence in order to fit the required number of units.

The above distinction about in-game AI is summarized in Figure 1.

AI in games in practice is often data-driven. Some scripting system is attached to the game engine in order to allow customization of AI behaviors. In their simplest form scripts are simply a (possibly very large) set of configuration parameters that customize the fixed units behavior. More advanced scripting systems [14, 17] allow coding detailed aspects of the characters behaviors in fully-fledged programming languages.

The authors goal is to create a data-driven AI system that is capable of expressing any point in the "scripting space" described above, that is they wish to be able to express scripts that are both complex to animate believable and smart characters and capable of doing so efficiently for hundreds or thousands of characters.

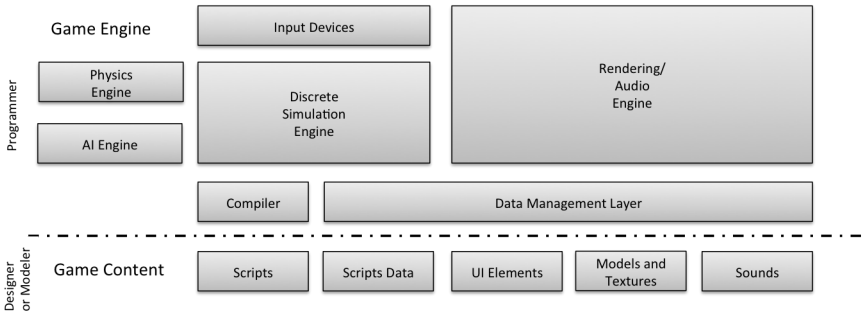

Figure 2: Data-driven game architecture

2 Data-driven games

A data-driven game [8] separates game content from the game code. This design allows both game programmers and game designers (groups with very different skillsets) to contribute separately to game development. This separation has long been practiced, for example by storing models, textures and sounds in data files separate from the game engine; this trend has recently accelerated by trying to move as much content as possible out of the engine: character and story-line data is increasingly [18] being stored in external configuration files. Modern design goes even further, storing game logic specific to game play in external files written in a scripting language such as Lua, Python or a custom-made scripting language [2, 7, 3].

In Figure 2 we can see the general architecture of a data-driven game. The game engine (AI, physics, rendering, audio and discrete simulation) is written and maintained by programmers. Typically, the discrete simulation engine puts together the interactions between physics, AI and game logic in a meaningful way, and exposes the resulting game state and events to the rendering and audio engine which give some feedback to the player. The game content is created by game designers, who are responsible with creating and populating the game world; the game world contains both artistic elements like models and sounds, but also logical elements such as behaviors and character statistics. Logical elements are defined by scripts; scripts are either compiled or interpreted and executed by the discrete simulation engine. Separating scripts becomes important in all those game where character behavior must be tested and adjusted constantly during development to ensure that there is no single optimal strategy that can ensure victory (otherwise the game would lose much of its challenge and interest). Also, separating game logic into scripts allows for "modding", that is players can modify the game even without access to its source code; the creativity of the players can extend the lifetime of a game beyond the original intentions of its developers ([15]). The entire focus of the authors is on the discrete simulation engine and how techniques taken from the database comunity

can be used to build a better performing implementation that can be heavily scripted.

The Discrete Simulation Engine Game are processed in clock ticks. During each clock tick the simulation engine processes the current state of the world, thereby computing its updated version. This usually consists in considering the actions of all the characters of the game (some characters may perform null actions, but still they are given an opportunity at every tick); each action may produce several *effects*. Effects produced during the same tick are written into the game state simultaneously, allowing us to cleanly separate each clock tick into three stages:

- a query stage when we read the contents of the game data
- a decision stage where we choose the actions of each character
- an update stage where we write into the game state the effects produced by each action

Since actions are all updating the game data simultaneously, we use a transaction model to describe how these updates are processed. Since effects usually increase or decrease numeric values then such a model becomes simply an aggregate function such as `(+)`, `(-)`, `max`, `min`, ... Effects are usually separated into *stackable* and *non-stackable*, depending on whether or not they accumulate during one tick or only one update is picked; for example, damage is stackable since a unit that suffers damage from multiple opponents receives the sum of all these damages, while healing auras are non-stackable because only the most beneficial aura is applied while all the others are discarded.

3 Real-time strategy games

Real-time strategy games are the ideal field of application of techniques that increase the number of supported characters, as demonstrated by the number of units available. In these games, the player controls large numbers of *units* which are selected and which receive commands; the execution of commands by each unit is controlled by the AI; the smarter the AI, the more entertaining the game: a player wants his units to follow his instructions correctly, without having to specify too many details; this way, when the player has issued orders to some units, he can move to other portions of the map and give other orders to other units while the original orders are being carried out. Unfortunately unit behavior in current RTSes is quite primitive, usually modeled with simple finite state machines. As a result, players need to issue many orders to achieve a good level of coordination between his units. Processing scripts that handle the correct behavior of each unit is very expensive at run-time, since each unit is typically processed separately. A typical solution to this problem is that of processing units in groups (centralized AI); with this technique each script

controls a large number of units, which then behave with good coordination. For example, in the game *Warcraft III*, the AI for each computer player is divided in two commanders: defence and attack. Centralized AI often has problems when trying to maintain actions over multiple fronts (more than the number of virtual commanders available), and it is of no help to the human player. Having smarter units would benefit both the computer player and the human player.

Note that centralized AI is just an ad-hoc form of set-at-a-time processing, explicitly implemented by the game designer who aggregates units knowing that they will perform a similar role. The authors propose to build a scripting language that makes grouping units behaviors together more systematic with a declarative language like SQL, paired with a set of optimizations appropriate for games.

The most important functionality of scripts will thus be implemented with a purely functional language with aggregate operations on sets. The case study will be a synthetic RTS with three types of units:

- armored **knights** who can move and attack at short range
- **archers** who can move and attack at long range
- **healers** who heal friendly units within a certain range; healing auras are nonstackable

4 SGL

Game data is modeled as a relation E. This table is a multiset and it needs not have keys. Each row in the table represents a unit or object, with information such as health, speed, damage, special properties, etc. Each row also contains data representing messages to this unit, such as data coming from the pathfinding system, cooldown periods, and so on. One possible definition of E for our example could be:

```
E(key,player,posx,posy,health,cooldown,
  weaponused,movevect_x,movevect_y,damage,inaura)
```

The attributes `key ... cooldown` represent the state of the unit; these attributes are not modified directly from a script, but rather they are modified accordingly to the values of `weaponused ... inaura`, which represent the effects that unit is being subjected to.

An SGL script consists of a single action for a single unit, and at each tick it will be executed and it will take an entire environment E and it will return a new environment E_u. All environments E_u are then combined into a single environment in which the effects are applied with a post-processing step. A possible post-processing related to the schema above could be (where `norm` normalizes movement velocity):

```
SELECT u.key, u.player,
```

```
       u.posx + u.movevect_x * norm AS posx,
       u.posy + u.movevect_y * norm AS posy,
       u.health - u.damage + u.inaura AS health,
       u.cooldown - 1 + u.weaponused*_TIME_RELOAD AS
          cooldown,
       0 AS weaponused,
       0 AS movevect_x,
       0 AS movevect_y,
       0 AS damage,
       0 AS inaura
FROM E u
WHERE u.health > 0
```

The post-processing step above applies the effects to the state and resets the effects to avoid applying them again during the next time step.

Syntax of SGL SGL scripts have a simple syntax consisting of a mix of ML (`let, if-then-else`) and SQL:

```
main(u) {
  (let c = CountEnemiesInRange(u,u.range))
  (let away_vector = (u.posx, u.posy) -
         CentroidOfEnemyUnits(u, u.range)) {
           if (c > u.morale) then
             perform MoveInDirection(u,away_vector);
           else if (c > 0 and u.cooldown = 0) then
             (let target_key = getNearestEnemy(u).key)
                 {
             perform FireAt(u, target_key);
} } }
```

Where the auxiliary functions described above can be defined as:

```
function CountEnemiesInRange(u, range)
returns SELECT Count( * )
        FROM    E
        WHERE E.x >= u.posx - range
        AND     E.x <= u.posx + range
        AND     E.y >= u.posy - range
        AND     E.y <= u.posy + range
        AND     E.player <> u.player;
```

In general, actions have the following syntax:

```
action ::= (let attributename = term)
         | action action; action
         | if cond then action
         | if cond then action else action
```

```
        | perform action_name
```

Combining effects Effects in an environment E must be combined somehow. To do so, we add annotations to each attribute in the environment schema:

$$E(K, A_1 : \tau_1, ..., A_n : \tau_n)$$

where τ_i is an aggregate function such as `min, max, avg, sum, ...` which defines the way multiple effects on each unit are combined. A combination operator $\oplus R$ is defined as:

```
select K, f_{i_1}(A_{i_1}) as A_{i_1},...,f_{i_m}(A_{i_m}) as A_{i_m}
from R group by K,A_{i_1},...,A_{i_l};
```

where $f_i(A_i)$ is defined as A_i if $\tau_i = const$ (that is no aggregation is needed on attribute i) while it is defined as τ_i otherwise (that is it uses the aggregate function described by τ_i itself). We can define (unambiguously) another meaning for $\oplus$: $R \oplus S = \oplus(R \uplus S)$ where $\uplus$ denotes multiset union.

In the case of the schema described above, the $\oplus E$ environment can be computed as:

```
SELECT key, player, posx, posy, health, cooldown,
       max(weaponused) AS weaponused,
       sum(movevect_x) AS movevect_x,
       sum(movevect_y) AS movevect_y,
       sum(damage) as damage,
       max(inaura) as inaura
FROM E
GROUP BY key, player, posx, posy, health, cooldown
```

Each action can be seen as a function from a unit and the environment into the environment which at each evaluation step applies the combination operator; actions have type:

$action : Env \times Multiset(Env) \rightarrow Multiset(Env)$

which is the denotational equivalent of a stateful function. The first parameter is the tuple representing the current unit, the second parameter is the set of all units, the result is the new set of all units. The semantics of SGL actions can be given in terms of a semantic function $[\![\bullet]\!]$:

```
[[let v = t in f]](u,E)     := [[f[v ↦ [[ t ]]](u,E)]](u,E)
[[f;g]](u,E)                := [[f]](u,E) ⊕ [[g]](u,E)
[[if c then f]](u,E)        := if [[c]]_cond(u,E) then
[[perform(G)]](u,E)         := [[G]](u,E)
```

where `G` is a built-in action function or a user-defined auxiliary function, and where `if-then-else` can easily be defined in terms of sequential conditionals with the same condition (negated in the `else` branch).

Optimization As with typical relational systems, there are two possible optimization venues: algebraic and physical. Algebraic optimizations are applied to queries in order to ensure that a script combines as often as possible in order to keep the number of elements to process to the minimum possible; for example, rather than compute:

$$\oplus(R \biguplus S)$$

it is usually convenient to compute:

$$\oplus(\oplus R \biguplus \oplus S)$$

thanks to the fact that the $\oplus$ operator returns a smaller set (effects are condensed together into the state).

More important in terms of performance gain is the use of indices.

Sometimes a unit must process an aggregate function for example to count the number of nearby units; for example, a unit may wish to flee if its allies are overwhelmed by a large enemy force. This script can be naïvely computed with an $O(n^2)$ algorithm. This computation can be greatly accelerated with the introduction of an index for the aggregate that allows to share the results of the aggregation across several units.

SGL queries have the advantage that the set of queries is fixed a priori, and thus the indices can be tailored around each individual query plan. Also, it is important to realize that while indices can be used to share information between nearby units, indices are discarded and rebuilt at the beginning of each clock tick beacuse of the very high dynamicity of unit data even across one single tick; for the most important and often updated indices such as those on unit positions, it is more efficient to do so than it is to maintain a dynamic index.

The use of indices reduces the complexity of many common AI algorithms from $O(n^2)$ to $O(n \log n)$. This gives games the possibility to scale to an order of magnitude more units without incurring in a significant performance hit.

5 Conclusions

Innovations in game architecture are often responsible for pushing further the boundaries of game design. By building more powerful *and* more scriptable game engines it is possible at the same time to create games where game logic is faster and more complex, resulting in a more compelling experience for the player.

Building a system that allows for a smarter, individualized AI could usher an era of games that offer a smarter experience where units and non-playing characters make smart choices rather than waiting for "obvious" instructions from the player.

It must also be noted that games have been historically responsible for many evolutions of modern personal computing, by pushing the hardware to its limits and by offering powerful visual experiences that years or decades later become

paradigms in non-gaming software. For this reason this kind of language evolution must be put in its right context: it is not just a way to make better games, but it is also a prototype for a way to write better *programs* in general.

Future research directions in the field From a survey of the current panorama of game development tools and languages it appears that a technological shift is happening: from the almost exclusive use of C and C++ for creating games, plus the odd use of Lua or Python as scripting languages, more and more games are being made in modern managed languages such as C# or Objective-C. The introduction of XNA (a framework that allows independent game developers to publish commercial games on the XBox 360 without costly licenses) has marked the beginning of the adoption of C#-based frameworks for making games; other similar frameworks have been used in games between a C++ engine and the scripting system. This shift signals that the game development industry may be starting to have trouble with its old infrastructure given the rising development costs, the shrinking budgets and the decreasing development time and it is looking into new languages with more abstraction to increase its efficiency. For this reason many researchers (the authors of this report being part of this group) are studying new ways to leverage the power coming from the field of functional and declarative languages in order to more easily automate some repetitive programming tasks that often require complex and cumbersome libraries.

References

[1] Entertainment software association. 2006 sales, demographic and usage data: Essential facts about the computer and video game industry. http://www.theesa.com/.

[2] Games using lua as a scripting language. http://en.wikipedia.org/wiki/Category:Lua-scripted_video_games.

[3] Unrealscript documentation. http://unreal.epicgames.com/UnrealScript.htm.

[4] AIAndy and Zalamander. Advanced melee ai in warcraft iii. http://www.ecs.soton.ac.uk/ lph105/AMAI.

[5] Robert Albright, Alan Demers, Johannes Gehrke, Nitin Gupta, Hooyeon Lee, Rick Keilty, Gregory Sadowski, Ben Sowell, and Walker White. Sgl: a scalable language for data-driven games. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1217–1222, New York, NY, USA, 2008. ACM.

[6] Alice Leung Brett Benyo David Diller, William Ferguson and Dennis Foley. Behavior modeling in commercial games. *In Proceedings of the Thirteenth Conference on Behavior Representation in Modeling and Simulation*, 2004.

[7] Bruce Dawson. Game scripting in python. http:// www.gamasutra.com/features/ 20020821/dawson_ pfv.htm, 2002. Game Developers Conference Proceedings.

[8] Mark DeLoura. *Game Programmer Gems 1.* Charles River Media, 2000.

[9] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. A functional i/o system or, fun for freshman kids. *SIGPLAN Not.*, 44:47–58, August 2009.

[10] Yolanda Garrido, Álvaro Marco, Joaquín Segura, Teresa Blanco, and Roberto Casas. Accessible Gaming through Mainstreaming Kinetic Controller. In *Intelligent Technologies for Interactive Entertainment*, volume 9 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, chapter 7, pages 68–77. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[11] John Laird and Michael van Lent. Interactive computer games: Human-level ai's killer application. *In National Conference on Artificial Intelligence (AAAI)*, 2000.

[12] Scott Leutenegger and Jeffrey Edgington. A games first approach to teaching introductory programming. In *Proceedings of the 38th SIGCSE technical symposium on Computer science education*, SIGCSE '07, pages 115–118, New York, NY, USA, 2007. ACM.

[13] Michael Mateas and Andrew Stern. Natural language understanding in faade: Surface-text processing. *In Technologies for Interactive Digital Storytelling and Entertainment (TIDSE)*, 2006.

[14] Don Moar and Jay Watamaniuk. *Introduction to the Aurora Neverwinter Toolset.* Bioware, 2006.

[15] Philip Rosedale and Cory Ondrejka. Enabling player-created online world with grid computing and streaming. Technical report, Gamasutra, September 2003.

[16] Sega. *Rome: Total War Manual*, 2004.

[17] Jake Simpson. Scripting and sims2: Coding the psychology of little people. *Game Developers Conference (GDC)*, 2005.

[18] Mustafa Thamer. Act of mod: Building sid meier's civilization iv for customization. *Game Developer*, August 2005.

[19] Greg Wadley, Martin Gibbs, Kevin Hew, and Connor Graham. Computer supported cooperative play, âĂIJthird placesâĂİ and online videogames. In *Interaction. University of Queensland: Brisbane*, pages 238–241, 2003.

[20] Walker White, Alan Demers, Christoph Koch, Johannes Gehrke, and Rajmohan Rajagopalan. Scaling games to epic proportions. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 31–42, New York, NY, USA, 2007. ACM.

[21] Walker White, Christoph Koch, Nitin Gupta, Johannes Gehrke, and Alan Demers. Database research opportunities in computer games. *SIGMOD Rec.*, 36:7–13, September 2007.

[22] Walker White, Benjamin Sowell, Johannes Gehrke, and Alan Demers. Declarative processing for computer games. In *In Proc. SIGGRAPH Sandbox Symposium*, 2008.