Designing Casanova: A Language for Games

Giuseppe Maggiore, Alvise Spanò, Renzo Orsini, Giulia Costantini, Michele Bugliesi, and Mohamed Abbadi

Università Ca’ Foscari Venezia
DAIS - Computer Science
{maggiore,spano,orsini,costantini,bugliesi,mabbadi}@dais.unive.it

Abstract. Games are complex pieces of software which give life to animated virtual worlds. Game developers carefully search the difficult balance between quality and efficiency in their games.

In this paper we present the Casanova language. This language allows the building of games with three important advantages when compared to traditional approaches: simplicity, safety, and performance. We will show how to rewrite an official sample of the XNA framework, resulting in a smaller source and a higher performance.

1 Introduction

Computer games promise to be the next frontier in entertainment, with game sales being comparable to movie and music sales in 2010 [5]. The unprecedented market prospects and potential for computer-game diffusion among end-users have 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 a complex business. Games are large pieces of software with many heterogeneous requirements, the two crucial being high quality and high performance [2]. 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. In contrast, simulation quality is often lacking in modern games; game entities often react to the player with little intelligence, input controllers are used in straightforward ways and the logic of game levels is more often than not completely linear. Building a high-quality simulation is rather 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 (and often even rebuilt from scratch) many times during the course of the development. For this reason game architectures require a large amount of flexibility.

To manage the complexity, game developers use a variety of strategies. They have used object-oriented architectures, component-based systems, and reactive programming, with some degree of success for this purpose [6,7,4].

In this paper we present the Casanova language, a language for making games, as a solution to the obstacles mentioned above. Casanova offers a mixed declarative/procedural style of programming which has been designed in order to facilitate game development. The basic idea of the language is to require from the developer only and exclusively those aspects of the game code which are specific to the game being developed. The language aims for simplicity and expressive power, and thanks to automated optimizations it is capable of generating code that is much faster than hand-written code and at no effort for the developer. The language offers primitives to cover the development of the game logic, and incorporates the typical processing of a game engine. Also, the language is built around a theoretical model of games with a “well-formedness” definition, in order to ensure that game code is always a good model of the simulated virtual world.

In the remainder of the paper we show the Casanova language in action. We begin with a description of the current state of game engines and game programming in Section 2. In Section 3 we define our model of games. We describe the Casanova language in Section 4. We show an example of Casanova in action, and also how we have rewritten the game logic of an official XNA sample from Microsoft [18] in Casanova with far less code and higher runtime performance in Section 5. In Section 6 we discuss our results and some future work.

2 Background

In this section we discuss five current approaches to game development. The two most common game engine architectures found in today’s commercial games are (1) object-oriented hierarchies and (2) 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 game engine [1]. A component-based system defines each game entity as a composition of components that provide reusable, specific functionality such as animation, movement, and reaction to physics. Component-based systems are being widely adopted, and they are described in [6].

These two, approaches are rather traditional and suffer from a noticeable shortcoming: they focus exclusively on representing single entities and their update operations in a dynamic, even composable way. 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.). Usually much of a game complexity comes from defining (and optimizing) these interactions. Moreover, 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.

There are two more approaches that have emerged in the last few years as possible alternatives to object-orientation and component-based systems. They are (3) (functional) reactive programming and (4) SQL-style declarative programming.