Prototyping a Game Engine Architecture as a Multi-Agent System

Carlos Marin
Institute of New Imaging Technologies
Universitat Jaume I
Spain, Castellón
cmarin@uji.es

Miguel Chover
Institute of New Imaging Technologies
Universitat Jaume I
Spain, Castellón
chover@uji.es

Jose M. Sotoca
Institute of New Imaging Technologies
Universitat Jaume I
Spain, Castellón
sotoca@uji.es

ABSTRACT

The game engines are one of the essential and daily used applications on the game development field. These applications are designed to assist in the creation of this type of contents. Nevertheless, their usage and development are very complex. Moreover, there are not many research papers about the game engine architecture definition and specification. In this sense, this work presents a methodology to specify a game engine defined as a multi-agent system. In such a way, from a multi-agent approximation, a game engine architecture can be prototyped in a fast way. Also, this implementation can be exported to a general programming language for maximum performance, facilitating the definition and comprehension of the mechanisms of the game engine.

Keywords
Game engine architecture, game development, multi-agent system.

1 INTRODUCTION

The game engines are frameworks composed by tools and interfaces that increase the abstraction level over low-level tasks to develop a video game [Gre14]. They are designed to simplify the creation of video games, encouraging the reuse of components and by abstracting the communication with the hardware and the operating system running the game [Nys14]. This method reduces development times since it provides tools to solve common issues on most of the games. Each game engine has its own components organisation structure known as architecture. This architecture determines the organisation of the modules composing the engine and the communications between them, the operating system and the hardware drivers.

Even though its goal is to ease game development, these applications are not easy to use. Game engine development is a complex task despite the reduced number of academic papers on game engine architecture [And08, Amp10]. The current literature deals with the engine components, such as the behaviour specification, the scene render or the networking. Nevertheless, the game engine architecture connecting all these is a subject that has been barely covered. In fact, [And08] brings forward the lack of academic papers and research lines on this matter. Some of these lines are the identification of the software components common to every kind of game, and the research of common elements to every kind of game engine in order to define a genre-independent reference architecture and the recognition of the best practices on game engine development.

Multi-agent systems (MAS) are adapted to the distributed problem resolution, where multiple modules work autonomously over a shared environment to achieve a goal. In this sense, the primary objective of this work is to demonstrate how MAS are capable to easily specify the system's components that define a game engine architecture. For this purpose, a game engine architecture prototype based on MAS has been designed, with autonomous components, so that they can perform specific tasks on the elements of the game. The proposed game engine must be able to run completely functional games from this agent-based architecture. To this end, the prototype has been developed on the MAS development platform *NetLogo* [Tis04].

The remainder of the work is organised as it follows: state of the art on game engine architecture and its relationship with MAS is presented in section two. Section three shows a summary of the principal MAS features. Subsequently, the game engine architecture proposed on this work is presented in section four. Finally, in section five, the results are shown and subsequently discussed in section six, and in section seven, the conclusions of this work are presented along with the future research lines.

2 STATE OF THE ART

This section presents a compilation of technical works that associate MAS with the game engine and their appearance on academic papers. Game engine architecture is defined as the organisation structure of the engine's components and their relationship with its supporting drivers, hardware and operating system [Gre14]. In a game engine, the components are the subsystems responsible for running specific tasks such as the rendering, the logic evaluation, the user interaction or the physics evaluation. Some of these subsystems can be considered as complete engines by themselves due to its complexity. In fact, the logic component is not easy to standardise due to its inherent connection with the mechanics of each game [Mil09, Lew02, Doh03, Amp10].

Among all the methods trying to specify the game engine components, the MAS is paradigmatic because it relies upon autonomous entities communicating with one another and performing tasks on a shared environment, and so it can easily relate the game and the behaviour specification elements [Nys14]. This cooperative distributed problem-solving fits perfectly with the task distribution on a game engine. Furthermore, the academic papers linking games and MAS becomes evident the implicit connection between these interactive systems and the agent-based systems in the following aspects: the game element's and the agent's concepts and their interrelation, their communication protocols and their cooperation mechanisms [Woo02, Gla06, Pos07, Sil03].

In [Pon13], the authors propose an agent-based system to control the game parameters according to the game objectives. Besides, [Fin08] shows a system where the agents learn autonomously to play games without human intervention, [Gra13] proposes a system to integrate virtual worlds with a multi-agent interface and [Jep10] proposes an agent creation framework for serious games.

Furthermore, [Dig09] shows the relationship between MAS and the game mechanics design, emphasising on the industry tendency to associate the game mechanics definition and the natural language on the game development. Moreover, [Bec14] develops a MAS based on the commercial game engine Unity, by doing a 3D simulation of the behaviours related to the path-finding methods for multiple agents on a passenger airport context.

Additionally, on the multiuser issue, [Ado01] presents a MAS to handle the multiuser mechanics on a tournament game, [Sac11] presents an intelligent agent-based distribution to build multiplayer systems and [Ara08, Ara12] introduces a MAS to conduct the design of Multiplayer Massive Online Games (MMOG), a specific type of games where the priority is the real-time interactivity of several game agents.

Lastly, [Gar06, Gar07, Gar10] shows a virtual environment where the agents are communicating with the player like in a 3D chat and then [Rem15] introduces an application specification for a 3D virtual fair as a MAS.

3 MAS FEATURES

According to the M. Wooldridge definition [Woo02], an agent is a computer system that is situated in an environment in order to meet its design objectives. Based on the general definition for a MAS, it is necessary to consider the following formal characteristics:

- The environment of the MAS can be in any of the discrete states of a finite set E = [e_0, e_1, e_2, ...] of states.
- The agents in the system have a set of available actions with the capacity to transform their environment Ac = [α_0 , α_1 , α_2 , ...].
- The run r of an agent on its environment is the interlayered sequence of actions and environment states r: $e_0 \rightarrow^{\alpha_0} e_1 \rightarrow^{\alpha_1} e_2 \rightarrow^{\alpha_2} \dots e_{u-1} \rightarrow^{\alpha_{u-1}} e_u$, where $R = [r, r', \dots]$ represents the set of all the possible and finite sequences of E and Ac.
- The effect of the agent actions on an environment comes determined by the transformation function τ: $R^{AC} \rightarrow \wp(E)$ [Fag95], where R^{AC} represents the subset of R ending on an action, and R^E represents the subset of R ending on an environment state.

In this sense, the following definitions can be established:

Definition 1: An environment *Env* is defined as a triple *Env* = <E, e_0, τ>, where E is a set of states, $e_0 \in E$ is the initial state and τ is the state transformation function.

Definition 2: An agent *Ag* is defined as a function *Ag*: $R^E \rightarrow Ac$ establishing a correspondence between runs and actions. It is assumed that these runs end on an environment state [Rus95].

In this sense, an agent selects the action to perform based on the system's history that it has witnessed. It is necessary to take into account that the environments are implicitly non-deterministic, but the agents are deterministic. The set of all the agents *Ag* of a system is represented as *AG*. The set of all the runs of an agent *Ag* on an environment *Env* is *R*(*Ag*, *Env*).

Definition 3: A purely reactive agent [Gen87] is defined as a function *Ag*: $E \rightarrow Ac$, which indicates that their decision-making process only with the current environment state.

Definition 4: An agent is considered as a perceptive agent when it is composed of a perception function and an action function. It is represented as *Ag* = <*see*, *action*>, where *see* is a function *see*: $E \rightarrow Per$ and *action* is a function *action*: $Per^* \rightarrow Ac$.

4 ARCHITECTURE PROTOTYPE

In order to define the game engine architecture, this study is focused on the engine's components definition and the behaviour specification for the game elements. In this sense, the formal correspondences with a MAS are established with the aim of proving that the agent-based engine is capable of making games. On this architecture, the MAS is the element that structures the information distribution and communication between the engine's elements. The prototype design is oriented towards the creation of 2D games, and it has been developed on the agent-based programming language *NetLogo* [Tis04].

Next, the elements of the game engine are presented from a MAS point of view:

- The engine environment, or the space of coexistence for every element of the game.
- The engine's agents, the elements equivalent to the engine components.
- The game's agents, also known as game objects or actors.

4.1 The engine environment

In this game engine, the environment *Env* represents the conjunction by the agents representing the engine's components or engine's agents and the game objects, also known as actors. This environment *Env* is in charge of storing the game's states *E* through its properties. Furthermore, starting from an initial state e_0, the modules perform transformations τ on the general properties of the environment and on the actors with which they have dependencies. The initial state e_0 determines the set of original properties present in the environment. Among the properties that define the state of the engine, there are the following categories:

- **Basic properties**: Properties related to the resolution of the output screen, the characteristics of the camera and the sound of the game.
- **Physic properties**: Characteristics that control the physical behaviour of the environment such as gravity or air friction.
- **Timers**: Set of properties responsible for controlling and providing information to the game elements on execution times, time differentials and elapsed time.
- **Lists**: The lists are responsible for maintaining the lists of actors used by each engine component, for storing the input events that have occurred in the engine and to manage the sounds queue.

The execution of the environment on the engine's architecture allows evaluating the processes related to the correct functioning of the game. The execution of the game $R(AG, Env)$, represents the application of the set *AG* for all agents of the engine respect to the environment *Env*. The process that relates the agents to the elements of the game is known as the game loop: the cycle of evaluation, updating and continuous execution of the game. It is responsible for managing the interaction between the engine's agents and actors; that is, it is the process through which the agents that represent the engine components apply their behaviour rules on the environment.

The engine agents' communication with the environment is not the same for all elements that compose it. Also, this engine agents work asynchronously and at different frequencies, however its sequential implementation is absolutely valid. Figure 1 shows a diagram describing the communications between the engine environment, the engine agents, and the actors, where the processes of information transmission are determined.

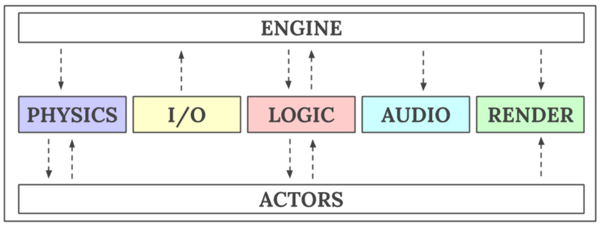

Figure 1: Communications' diagram between the components of the game engine.

4.2 The engine's agents

The engine agents that perform the role of the game engine components are the *Physics agent*, the *Input agent*, the *Logic agent*, the *Audio agent* and the *Render agent* (see Figure 1). These agents represent autonomous components that execute actions α on the environment *Env*. An agent *Ag* is an element that acts from the environment state *e* and its internal state, including the tasks assigned to it. In the designed architecture, engine agents represent the components of the game engine. Their behaviour is determined by predefined behaviour rules that evaluate the environment properties and the game actors, and which determine the actions to be taken. These agents are responsible for executing the actions *Ac* on the environment *Env*.

The engine agents present a reactive behaviour since they carry out their decision-making processes taking into account the present state of the environment and the environment's history that they have witnessed. Moreover, the *Input agent* and the *Audio agent* present a perceptual behaviour, and some of their essential functions are to wait until the perception function registers events that they can interpret. The different game engine agents are described below:

- **Physics agent**: Responsible for evaluating the actor's physical behaviour according to the environment properties and its configuration. The execution of this agent requires a high-frequency cyclic performance due to the need to iterate the numerical integration of the equations repeatedly. Also, this agent executes actions on the environment in charge of detecting collisions, applying the response to the collision, integrating the movement equations and updating the position of the actors.

- **Input agent**: Manages the communication between the engine and the user. It transforms input events into interpretable information for the engine. It has an asynchronous behaviour that stores its input data into the event list of the engine.

- **Logic agent**: Observes the environment state and runs the actor's behaviours rules so that they fulfil their tasks. It is a cyclic operation, with lower latency than the physics agent one.

- **Audio agent**: Reproduces sounds asynchronously while the sound queue is not empty.

- **Render agent**: Draws the environment state on the screen in a repeatedly with sufficient latency to meet the threshold of human vision.

The communication between engine agents
Next, a brief description of how the communication is established between the agents of the engine with the environment and the game actors:

- The *Physics agent* must access the properties of the actors to apply the transformations resulting from the evaluation of their physical behaviour. For these evaluations, it is relying on the physical set-up of the game, stored within its properties.

- The *Input agent* obtains the information from the events that are given from the operating system. It notifies the engine, with the aim that other engine agents, such as the *Logic agent*, can access this information.

- The *Logic agent* is the only agent with the ability to read and write both in the environment properties and in the general properties of the actors. This feature is necessary in order to set and to get all the necessary information for any required game mechanics.

- The *Audio agent* works on its reproduction queue. This queue is loaded by the *Logic agent* when it is required to play a sound.

- The *Render agent*, for its correct operation, requires information about the environment with the resolution of the screen or the properties of the camera as well as the actors, with their textures, dimensions and transformations.

4.3 The game's agents

The game actors are the agents that make up the games: characters, scenarios and props; and are in charge of performing the game mechanics. All of them have common characteristics and include a set of properties that determine their appearance and behaviour. The actors have the following types of properties:

- **Basic properties**: These properties are related to the position, rotation and scale of the actors.

- **Rendering properties**: The properties connected to the image representing the game actor and its visualisation. Also, it can display text defined by a font, a size, a colour and a style property.

- **Physics properties**: Properties that allow defining the physical characteristics of the actors such as speed, angular velocity, material properties (density, friction, restitution).

- **New properties**: Also, the actors can define new properties that expand their property list.

The *Logic agent* controls the actor's behaviour. Each game mechanic is determined by a set of behaviour rules stored in the actors. These behaviour rules are defined by logic statements, which in this case, are taking the form of a script. In the *NetLogo* case, they are implemented as functions.

5 THE USE CASE AND RESULTS

After the definition of the architecture, a use case is presented below to test the capabilities of the engine. The implementation is based on the classic arcade game *Arkanoid*, where the user must manage a paddle to control the bounces of the ball and destroy the most significant number of bricks without letting the ball colliding with the lower limit of the screen. The selection of this game as a demonstrator is because it includes features that require all engine components. The implementation of this example requires engine agents controlling the execution of the game and actor agents to perform the mechanics of the game. Nevertheless, the following behaviour algorithms are adapted to the case of use in order to simplify the presentation. Additionally, as stated above, there is only one type of actor agent: all the *Arkanoid* game elements are actors with configured properties and behaviour rules to fulfil their specific tasks.

For the game development, a *Paddle actor*, a *Ball actor* and four sets of fourteen *Brick actors* have been created. Also, there is an additional actor whose task is only to store and to show the game score as a new property. Initially, the *Brick actors* are distributed in a grid at the top of the screen, the *Ball actor* is initialised over the *Paddle* with a constant speed on the y-coordinate and a random velocity on the x-coordinate, and the *Paddle actor* is centred on the bottom. A capture of this layout is represented in Figure 2.

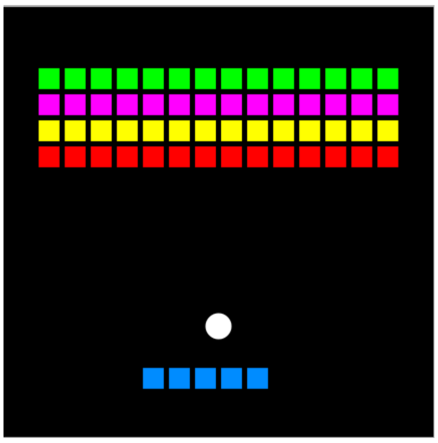
Figure 2: Capture of the *Arkanoid* game implemented with the prototyped engine.

Algorithm 1: Declaration of the environment properties, engine agents and actors agents in *NetLogo*.

```
globals [delta-time previous-time current-time]
physic-own [physic-list gravity-x gravity-y
  air-friction ]
input-own [event-list]
logic-own [logic-list]
audio-own [sound-list]
render-own [render-list resolution-x resolution-y
  camera-x camera-y camera-rotation camera-zoom]
actors-own [velocity-x velocity-y restitution
  active-physics? active-logic? active-audio?
  active-render? static?]
```

The declaration of these properties is presented in Algorithm 1, along with the engine agents properties and the actor agent properties. This property set can accept new properties depending on the game requirements. Moreover, during the game cycle execution, each engine agent evaluates the state of the environment and executes its actions according to the tasks it has programmed (see Algorithm 2).

The first step on the game loop is carried out by the *Physics agent* (see Algorithm 3). The execution of its functions begins with the *collision-detection* function, which is responsible for detecting collisions between actors. Next, the *collision-response* function is responsible for resolving the collision between two actors, which applies only to the actors with the active-physics property active. If any of these actors have a static physical behaviour, the response to the collision will not affect them. Finally, the *motion-integration* function is in charge of applying the motion equations to the actors. These movements are determined by the initial speed settings, the effect of gravity, friction with the air and the results of collisions. Its scope is about the actors with physical properties, as long as they are not static.

Algorithm 2: The game engine's game loop.

```
to go
  physic [ run-physic ]
  input [ run-input ]
  logic [ run-logic ]
  audio [ run-audio ]
  render [ run-render ]
end
```

Algorithm 3: Physics agent behaviour loop.

```
to collision-detection
  ask actors [
    if (distance ("ballID")) < size [ set colliding true
  ]
  ]
end
to collision-response
  ask actors [
    if (colliding and active-physics? and (not
  static?)) [
      set velocity-y (velocity-y * -1 * restitution) ]
    ]
  ]
end
to motion-integration
  ask actors [
    if (active-physics? and (not static?)) [
      set velocity-x (velocity-x + gravity-x *
  delta-time)
      set velocity-y (velocity-y + gravity-y *
  delta-time)
      set xcor (xcor + velocity-x)
      set ycor (ycor + velocity-y)
    ]
  ]
end
```

ISBN 978-80-86943-38-1

Upon the *Physic agent* completion, the *Input agent* behaviour for this use case is presented in Algorithm 4. This behaviour is based on the *event-management* function, the function responsible for managing the procedures that determine if an input event has occurred in the engine. When an occurrence is detected, the list of events of the environment properties is modified to communicate that information to the rest of the environmental elements.

Algorithm 4: Input agent behaviour loop.

```
to event-management
  if left-key [ set left-key-event true ]
  if right-key [ set right-key-event true ]
end
```

Next, the environment is evaluated by the *Logic agent*. In the general case, this procedure is responsible for evaluating the logic of each actor agent in terms of game mechanics. For this use case, two rules are presented to demonstrate the running of its behaviour. Algorithm 5 shows two rules based on the function *evaluate-actors*.

The first function controls the behaviour of the *Brick actor*. After the detection of a collision event involving the *Ball actor* and a *Brick actor*, the *Ball actor* gives the response to the collision to simulate the rebound, and the *Brick actor* in question is eliminated from the game. Additionally, the property points of the *Score actor* is increased by one, and a sound is added to the playback queue. Conversely, the second function sets the movement of the *Paddle actor* after the user inputs. Keyboard events control the movement of the *Paddle actor*: it applies a shift to the left and a shift to the right with the *A* and *Z* keys respectively. This displacement will occur as long as it does not reach the limits of the screen.

After adding a sound to the playlist, the *Audio agent* is responsible for its reproduction. The *Audio agent* behaviour cycle is presented in Algorithm 6, where the *play-sounds* function sets the audio playback based on the reproduction list information.

Finally, the last step on the game loop is performed by the *Render agent*. This agent executes the procedures of drawing the scene on the group of visible actors. Nevertheless, the rendering functions in *NetLogo* are minimal. For this example, the function *draw-actors* is composed just by a refresh call with the native function *tick*.

On a separate issue, the end of the game is reached when the *Ball actor* has no more *Brick actors* left to destroy or when he reaches the lower limit of the screen, that is when the player handling the *Paddle actor* is not able to return the *Ball actor* to the *Bricks* zone.

Algorithm 5: Logic agent behaviour loop.

```
; behaviour rule for the Brick actor
to evaluate-actors
  ask actors [
    ifelse (static? and colliding) [ die ] [ set points
   points + 1 ]
  ]
end
; behaviour rule for the Paddle actor
to evaluate-actors
  if left-key-pressed [
    ask (actor "paddleID") [
      if xcor > min-pxcor + size [
        ask actor "paddleID" [ set xcor xcor - size ]
      ]
    ]
  ]
  if right-key-pressed [
    ask (actor "paddleID") [
      if xcor < max-pxcor - size [
        ask actor "paddleID" [ set xcor xcor + size ]
      ]
    ]
  ]
end
```

Algorithm 6: Audio agent behaviour loop.

```
to play-sound
  ask actors [
    if empty? sound-list [
      sound:play-note "Gunshot" 60 64 0.25
    ]
  ]
end
```

It can be seen that the programming language for MAS *NetLogo* is not prepared to make aesthetically attractive games. For example, it is not possible to add new images or new sounds to agents; it only works with predefined elements. Also, it is not possible to scale these forms in a single direction, which significantly limits the possibilities to the game aesthetics. In any case, it easily allows creating a game engine prototype that can be implemented on a more suitable programming language.

6 CONCLUSIONS AND FUTURE WORK

The design of the game engine architecture presents a methodology to specify a game engine defined as a MAS but also has some characteristics that are interesting to analyse. In the first place, it should be noted that the autonomous behaviour of the engine components over a shared space makes possible the resolution of many problems inherent in the development of games. Since each of them has autonomous tasks that could be done without external intervention. Besides, the engine is enriched with the essential characteristics to create a wide variety of games, where some fancy features are no longer necessary for the conventional 2D game engine.

Conversely, it should be noted that games are played with a single type of actor agent and, there are no hierarchical relationships between them. All the elements that define a game: the markers, the player, the NPC; they have the same properties and the same behaviour rules system. Furthermore, it is not necessary to define a scene graph, which simplifies the internal architecture of the engine and the design of the games. There are no complex data structures such as vectors or matrices that are not necessary for the creation of most arcade games.

In summary, the objective of this work aims to define a simple architecture prototype that is capable of running a game. It starts from the formal definition of the MAS with the end of leading a definition of its essential elements. Besides, the study of game engines and their relationship with MAS has allowed to generate a broad knowledge about the architecture of commercial game engines and to establish a game engine specification intuitively and closer to the way of describing systems with natural language. The engine is the environment of the MAS, while the components are the agents of the engine and the actors are the agents of the game. From the environment state, agents can perceive information and react to certain states based on their predefined tasks. The behaviour associated with the tasks is determined by behaviour rules and a pre-established set of actions.

About the future works, this prototype has led to the creation of a game engine following the architecture designed on this work. It is being built over the programming language *JavaScript* in order to execute games on web browsers.

7 ACKNOWLEDGMENTS

This work has been supported by the Ministry of Science and Technology (TIN2016- 75866-C3-1-R) and the Universitat Jaume I research project (UJI-B2018-56).

8 REFERENCES

[Gre14] Gregory, J. (2014). Game engine architecture. AK Peters/CRC Press.

[Nys14] Nystrom, R. (2014). Game programming patterns. Genever Benning.

[And08] Anderson, E. F., Engel, S., McLoughlin, L., Comninos, P. (2008). The case for research in game engine architecture.

[Amp10] Ampatzoglou, A., Stamelos, I. (2010). Software engineering research for computer games: A systematic review. Information and Software Technology, 52(9), 888-901.

[Tis04] Tisue, S., Wilensky, U. (2004, May). Netlogo: A simple environment for modeling complexity. In International conference on complex systems (Vol. 21, pp. 16-21).

[Mil09] Millington, I., Funge, J. (2009). Artificial intelligence for games. CRC Press.

[Lew02] Lewis, M., Jacobson, J. (2002). Game engines. Communications of the ACM, 45(1), 27.

[Doh03] Doherty, M. (2003). A software architecture for games. University of the Pacific Department of Computer Science Research and Project Journal (RAPJ), 1(1).

[Woo02] Wooldridge, M. (2002). An introduction to multiagent systems. John Wiley Sons.

[Gla06] Glavic, M. (2006). Agents and multi-agent systems: a short introduction for power engineers.

[Pos07] Poslad, S. (2007). Specifying protocols for multi-agent systems interaction. ACM Transactions on Autonomous and Adaptive Systems (TAAS), 2(4), 15.

[Sil03] Silva, C. T., Castro, J., Tedesco, P. A. (2003). Requirements for Multi-Agent Systems. WER, 2003, 198-212.

[Pon13] Pons, L., Bernon, C. (2013, October). A Multi-Agent System for Autonomous Control of Game Parameters. In Systems, Man, and Cybernetics (SMC), 2013 IEEE International Conference on (pp. 583-588). IEEE.

[Fin08] Finnsson, H., Bjornsson, Y. (2008, July). Simulation-Based Approach to General Game Playing. In Aaai (Vol. 8, pp. 259-264).

[Gra13] Grant, M., Sandeep, V., Fuhua, L. (2013, November). Integrating Multiagent Systems into Virtual Worlds. In 3rd International Conference on Multimedia Technology (ICMT-13). Atlantis Press.

[Jep10] P. Jepp, M. Fradinho and J. M. Pereira, "An Agent Framework for a Modular Serious Game," 2010 Second International Conference on Games and Virtual Worlds for Serious Applications, Braga, 2010, pp. 19-26.

[Dig09] Dignum, F., Westra, J., van Doesburg, W. A., Harbers, M. (2009). Games and agents: Designing intelligent gameplay. International Journal of Computer Games Technology, 2009.

[Bec14] Becker-Asano, C., Ruzzoli, F., Holscher, C., Nebel, B. (2014). A multi-agent system based on unity 4 for virtual perception and wayfinding. Transportation Research Procedia, 2, 452-455.

[Ado01] Adobbati, R., Marshall, A. N., Scholer, A., Tejada, S., Kaminka, G. A., Schaffer, S., Sollitto, C. (2001, May). Gamebots: A 3d virtual world test-bed for multi-agent research. In Proceedings of the second international workshop on Infrastructure for Agents, MAS, and Scalable MAS (Vol. 5, p. 6). Montreal, Canada.

[Sac11] Sacerdotianu, G., Ilie, S., Badica, C. (2011, September). Software Framework for Agent-Based Games and Simulations. In 2011 13th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (pp. 381-388). IEEE.

[Ara08] Aranda, G., Carrascosa, C., Botti, V. (2008, September). Characterizing massively multiplayer online games as multi-agent systems. In International Workshop on Hybrid Artificial Intelligence Systems (pp. 507-514). Springer, Berlin, Heidelberg.

[Ara12] Aranda, G., Trescak, T., Esteva, M., Rodriguez, I., Carrascosa, C. (2012). Massively multiplayer online games developed with agents. In Transactions on Edutainment VII (pp. 129-138). Springer, Berlin, Heidelberg.

[Gar06] Garces, A., Quiros, R., Chover, M., Camahort, E. (2006). Implementing moderately open agent-based systems. In IADIS International Conference WWW/Internet 2006 (pp. 360-369).

[Gar07] Garces, A., Quiros, R., Chover, M., Huerta, J., Camahort, E. (2007, February). A development methodology for moderately open multi-agent systems. In Proceedings of the 25th conference on IASTED International Multi-Conference: Software Engineering, SE (Vol. 7, pp. 37-42).

[Gar10] Garces, A., Quiros, R., Chover, M., Camahort, E. (2010). Implementing virtual agents: a HABA-based approach. Int J Multimed Appl, 2, 1-15.

[Rem15] Remolar, I., Garces, A., Rebollo, C., Chover, M., Quiros, R., Gumbau, J. (2015). Developing a virtual trade fair using an agent-oriented approach. Multimedia Tools and Applications, 74(13), 4561-4582.

[Fag95] Fagin, R., Halpern, J. Y., Moses, Y., Vardi, M. Y. (1995). Reasoning about knowledge MIT Press. Cambridge, MA, London, England.

[Rus95] Russell, S. J., Subramanian, D. (1994). Provably bounded-optimal agents. Journal of Artificial Intelligence Research, 2, 575-609.

[Gen87] Genesereth, M. R., Nilsson, N. J. (1987). Logical foundations of artificial. Intelligence. Morgan Kaufmann, 2.