# A Game Engine To Make Games As Multi-agent Systems

Carlos Marín [a, c] · Miguel Chover [a] · José M. Sotoca [a] · Luis A. García [b]

[a] *Institute of New Imaging Technologies - Universitat Jaume I. Castellón. Spain.*

[b] *Computer Science and Engineering Department - Universitat Jaume I. Castellón. Spain.*

[c] Corresponding author at *Universitat Jaume I, Avda. Sos Baynat, s/n, Espaitec 2 (5th floor), Castellón 12071; Spain.*
*Email addresses: cmarin@uji.es (C.Marín), chover@uji.es (M.Chover), sotoca@uji.es (J.M.Sotoca), garcial@uji.es (L.A.García)*

## Abstract

Video games are applications that present design patterns that resemble multi-agent systems. Game objects or actors are like autonomous agents that interact with each other to describe complex systems. The purpose of this work is to develop a game engine to build games as multi-agent systems. The actors or game engine agents have a set of properties and behaviour rules with the end to interact with the environment of the game. The behaviour definition is established through a formal semantic based on predicate logic. The proposed engine tries to fulfil the basic requirements of the multi-agent systems, by adjusting the characteristics of the system, without affecting its potential. Finally, a set of games are introduced to validate the operation and possibilities of the engine.

## 1. Introduction

Game engines are tools to facilitate video game development. They were conceived to generalise and reuse properties, methods and procedures common to the majority of games [1]. Through them, designers can generate different game mechanics using the same components and scripts. Currently, the vast majority of commercial game engines are designed with equivalent organisational paradigms and similar software specification patterns [2], increasing their capabilities and performance. Nevertheless, different works show the need to establish a standard specification of the game engine architectures [3, 4].

When analysing the games produced by these engines, it is perceived that the design structures resemble the modelling patterns of multi-agent systems (MAS) [5]. The study of these game engines and their characteristics suggest that there is a close relationship between the concepts of game and MAS. This is appreciated in the game elements of the agents' definition [6], their relations [7] and its communication protocols and cooperation mechanisms [8]. In fact, there are several related topics, paradigms or applications where the MAS are used for the control of automatic processes and dynamic systems [9], or mechanisms of cooperation and consensus [10].

With all this, a game engine can be formally defined from the knowledge of a MAS. The initial hypothesis is that a game can be defined and specified as a system of agents interacting with each other. In this regard, the proposed game engine must be able to generate functional games that satisfy the properties of MAS. For this purpose, this work proposes a game engine for the creation of games defined as MAS. The engine's formalisation follows the mathematical notation stated by M. Wooldridge [11] in order to define the structure of the game engine and to fulfil with the requirements of the proposed system.

The manuscript is organised as follows. Section 2 presents the current state of the art on game engines and their relation with MAS. In section 3, the formal definition of a MAS is introduced. Later in section 4, the proposed game engine is defined by following the formal specification shown in the previous section, along with the behaviour definition system for the game engine agents. In section 5, the concepts stated in the previous section and its implications are discussed. In section 6, a series of use cases are presented to demonstrate the potential of the tool. Finally, in section 7, several conclusions and the possible improvements of the proposed game engine are presented.

## 2. Game Engines and Multi-agent Systems

The term game engine appeared in the mid-1990s in reference to the architecture of independent software components that defined video games such as *Doom* from *IdSoftware* [1]. This architecture gained value as game developers began to create generic modules that facilitate the creation of new games, and thus reduce development times. In a game engine, the modules are the subsystems responsible for executing specific tasks, such as the drawing, the user interactions or the game physics. However, some of these modules are not easy to standardise, either because they involve complex systems that can be considered as complete engines by themselves, or because of their embedded relationships with the mechanics of each game [12, 13]. The module responsible for managing the game mechanics is the logic module and the decision-making system. Its function is implicitly related to the character's internal processes associated with their autonomous potential actions [14]. The range of available actions is dependent on game mechanics and must be established by the game designer. A game object can have simple rules to determine what to do next, such as the case in which a Non-Playable Character (NPC) follows the player character as long as it is within a range of action.

These autonomous behaviours are directly dependent on the communications between the entities of the game, either sending information about the game properties or about the game objects properties. It is at this point where the relationship between games and MAS becomes evident. There are several cases in the literature where different perspectives address the relationship between game concept and MAS. Thus, some present the relationship between MASs and the game mechanics design, emphasising the industry's tendency to relate natural language and the game mechanics definition during the creation of games [15], while others propose a framework for the creation of agents for serious games [16]. Further works present a MAS to model architecture for Massive Multiplayer Online Game (MMOG) games [17, 18], where the real-time interactivity of multiple game agents prevails. Other authors show a system that tries to integrate virtual worlds with a multi-agent platform through an interface [8]. Also, the MASs have been used to expose a virtual environment where agents communicate autonomously with the player as a 3D chat [19, 20, 21], and subsequently it was extended for the implementation of a virtual fair [22, 23, 24]. In addition, they have been employed to raise a distribution where it is possible to build multi-player systems based on intelligent agents [25], to propose a MAS to manage multi-user mechanics in a tournament game [26], and to present a system based on agents that control the parameters of the game according to the objectives to be achieved [27]. Finally, in terms of the relationship between MAS and game engines, a MAS based on the Unity game engine has been developed [28], generating a three-dimensional search behaviour simulation of multiple agents in the context of a passenger airport [29], and also a system where agents learn autonomously to play multiple games without human intervention [30].

Nevertheless, the approaches described above add MAS features to specific games or specific genres, but no one of them introduces a functional approach of MAS as the core of the game development. In this sense, the MAS would fulfil one of the original purposes of the game engines: the flexibility in the procedures and the modules aggregation.

## 3. Multi-agent Systems Features

Following the agent definition proposed by M. Wooldridge [11], an agent is a computer system that is situated in an environment, and it can perform autonomous actions in this environment in order to meet its design objectives. More specifically, in the general definition of MAS, it is necessary to take into account specific features when carrying out its formalisation:

- The environment to which the agents belong can be in any of the discrete states of a finite set of states $E = [e_0, e_1, e_2, ...]$.
- The agents have a set of possible actions available with the ability to transform their environment $Ac = [\alpha_0, \alpha_1, \alpha_2, ...]$.
- The run $r$ of an agent on its environment is the interlayered sequence of actions and environment states $r$: $e_0 \rightarrow^{\alpha 0} e_1 \rightarrow^{\alpha 1} e_2 \rightarrow^{\alpha 2} ... e_{u-1} \rightarrow^{\alpha u-1} e_u$.
- The set of all possible runs is $R$, where $R^{Ac}$ represents the subset of $R$ that ends with an action, and $R^E$ represents the subset of $R$ that ends with a state of the environment. The members of $R$ are represented as $R = [r_0, r_1, ...]$.
- The state transformation function $\tau$ introduces the effect of the actions of an agent on an environment $\tau$: $R^{Ac} \rightarrow \wp(E)$ [31].

Thus, the following definitions can be established:

**Definition 1.** An *Env* environment is defined as a triplet $Env = \langle E, e_0, \tau \rangle$, where $E$ is a set of states, $e_0 \in E$ is an initial state and $\tau$ is the state transformation function.

**Definition 2.** An *Ag* agent is defined as a function *Ag*: $R^E \rightarrow$ Ac and establishes a correspondence between runs and actions. It is assumed that these runs end in an environment state [32].

In this way, an agent makes the decision on what action to take based on the history of the system it has witnessed to date. The set of all agents *Ag* in a system is represented as AG and the set of runs of an agent *Ag* over the environment *Env* is represented as R(*Ag*, *Env*).

**Definition 3.** A purely reactive agent is defined as a function *Ag*: E → Ac [33], which indicates that they make the decision based only on the present state of the environment.

**Definition 4.** An agent with perception is considered as such when it is composed of perception functions and actions, as *Ag* = <*see*, *action*>. Where *see*: E → *Per* maps environment states to percepts and *action*: $Per^* \rightarrow$ Ac maps sequences of percepts to actions.

**Definition 5.** Agents with an internal state are those that have an internal data structure I used to store information, where I = [$i_0$, $i_1$, $i_2$, ...] is the set of all internal states of an agent. In this case, the *action* function is defined as a correspondence *action*: I → Ac. Additionally, there is a *next* function that generates new internal states from perceptions *next*: I x *Per* → I. The action to be performed by the agent will, therefore, be *action*(*next*(*i*, *see*(*e*))). After completing the action, the agent re-enters the perception cycle of the environment, it updates its status through *next*, and it selects the action to be taken with *action*.

Agents are goal-oriented, as they perform actions to satisfy some goal. The way these goals are represented is by using task predicates Ψ.

**Definition 6.** Let Ψ(*r*) indicates that the run $r \in R$ satisfies the predicate Ψ as $R_\Psi$(*Ag*, *Env*) = { *r* | *r* ∈ R(*Ag*, *Env*) and Ψ(*r*) }. Then, an agent *Ag* succeeds in the task <*Env*, Ψ> if $R_\Psi$(*Ag*, *Env*) = R(*Ag*, *Env*). In other words, *Ag* succeeds in <*Env*, Ψ> if every run of *Ag* in *Env* satisfies Ψ. Thereby, an agent just runs its right tasks.

## 4. The Game Engine

To define the structure of the proposed game engine, this work focuses on the definition of the elements that make up the system and the behaviour specification model generated for these elements. In this sense, formal correspondences are established between the games and MAS, to demonstrate that the engine allows specifying games defined as a MAS. The presented game engine makes possible the creation of 2D video games.

### 4.1. The Game

The engine allows defining the environment of the game where the action is performed. The run of the game *R*(AG, *Env*), represents the run of all agents AG on an environment *Env*. This environment *Env* is responsible for storing the game states E through the properties of it. The initial state $e_0$ determines the set of initial properties. The properties defining the game state are:

- **Camera**. It includes the camera position, rotation and zoom on the environment where the action takes place.
- **Audio**. A set of parameters to define and modulate the game's sounds such as pan, volume, start time and loop.
- **Physics**. They define, for instance, the intensity of the gravity force on the game, its influence on the actors and their physical properties. The game engine, therefore, is able to build games with realistic physics.

The agents AG of the system are described below, and they are called actors. These actors can transform the environment states according to their tasks and through the transformation operator $\tau$.

### 4.2. The Actor

The actors are responsible for running the actions Ac on the environment *Env*. In the proposed engine, the actors are purely reactive agents [33], since they make the decision-making process with consideration of the present. Also, the actors are agents with perception [11, 32], since in every moment they are watching the game state to evaluate it and to act accordingly. The actors have internal state $I = [i_0, i_1, i_2, ...]$, which is initially defined with a set of properties. The basic properties of the actors are classified into the following categories:

- **Geometry.** It includes properties related to the position, rotation and scale of the actors.
- **Render**. In this case, it includes the actor image and its related properties: opacity, flip, scroll, tile and tinting colour.
- **Text**. The actors can write text on the screen, according to a font, size, colour and style parameters. They can also write the value of any property from the game or from any actor.
- **Audio.** The specific sound the actor plays and the set up of its properties: start time, volume, pan and loop.
- **Physics.** The physical features for the actors such as speed, angular velocity and material properties: density, friction and restitution.
- **New**. Also, the actors can incorporate new knowledge as new properties that expand their internal state.

In the game engine, the perception of the environment is defined by the evaluation of the actor's physical behaviour and by the signals derived from the interaction with the user. Based on the actor's state and the perception of the environment, it is produced an evaluation of the actor's knowledge, generating a change in the game state or in the actors' state that allows selecting the actions $\alpha$ associated with its tasks. Logical predicates $\Psi$ define the specification of the tasks performed by the actors. The actors run their tasks in the system if $\exists r \in R(Ag, Env)$ such that the predicate $\Psi(r)$ is satisfied. Figure 1 shows the interaction cycle between the actors and the game.

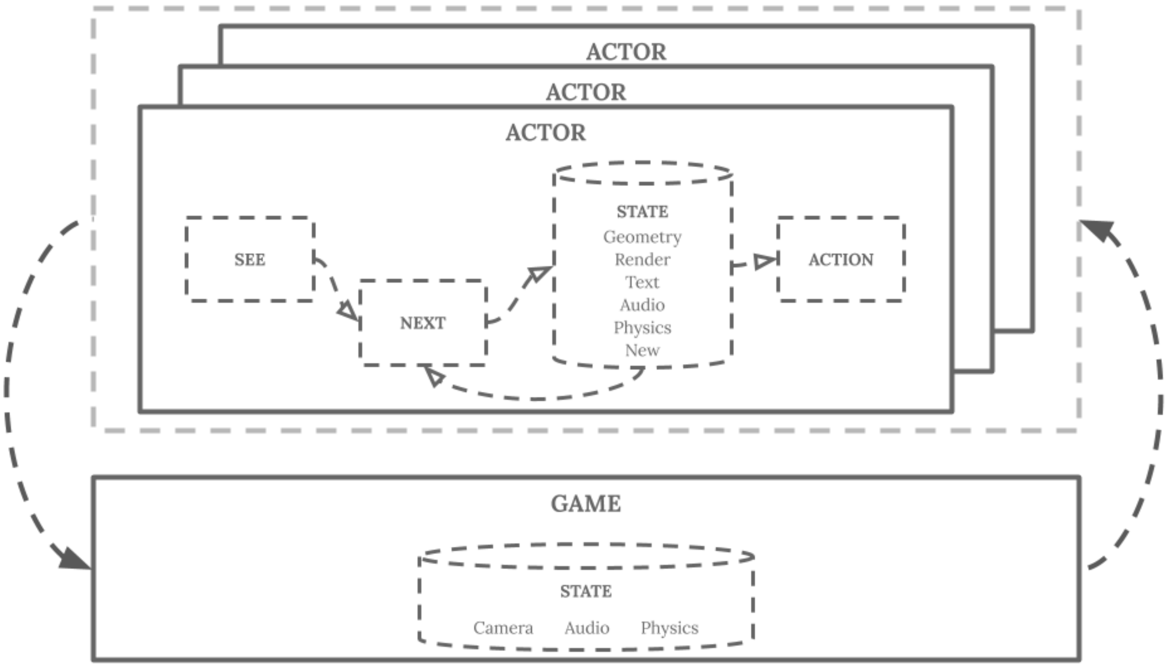

*Figure 1 .- The actors and their interaction cycle with the game.*

### 4.3. Behaviour Specification

The behaviour specification system is responsible for telling the actors what tasks to perform, based on the game state and the internal state of the actors, following a predefined semantic. This semantic is defined by a syntax of logical and non-logical symbols. Logical symbols include logical connectives and variables while non-logical symbols include predicates and functions [34]. The domain for this interpretation is the set of all the actors and the game.

The predicates $\Psi$ defined in first-order logic specify the actor's behaviour rules. In this way, the tasks that an actor might perform are organised into predicate formulas

$$\{ \Psi_0 \wedge \Psi_1 \wedge \Psi_2 \wedge \dots \}$$

where each formula can contain the following predicate structures:

- **Condition structure**: It is modelled by a predicate sequence model based on the structure of the IF-THEN-ELSE rules [35].

$$( \mathit{If} \rightarrow \varphi ) \wedge ( \neg \mathit{If} \rightarrow \theta )$$

where, *If* is a conditional literal predicate and both φ and θ are sequences of new predicates to be evaluated if the condition is met or if it is not met, respectively.

- **Action structure:** It is composed of an atomic element that includes a single literal predicate *Do*.

**Conditional Predicate If**

The conditional predicate represents the evaluation element of a condition in the decision-making process. In this game engine, the evaluation of the condition is based on a function that assesses the relationship between system entities.

*If( function( parameters ) )*

The parameters can contain variables as arithmetic expressions, which can include game or actor properties, mathematical functions and constant numerical values.

This function can be of the following types:

- **Compare**: This function compares numerical and boolean values. The relationships are established by a set of comparison operators whose elements can be *greater*, *greater or equal*, *equal*, *less* or *equal and less*.

  $compare(\ x, y, label\ )\ x, y \in \mathbb{R}, label \in [greater, greaterEqual, equal, lessEqual, less]$

- **Timer**: This function handles the system's timing. It determines if a specific time $x$ has expired so that the evaluation of the function is true.

  $timer(\ x\ ), x \in \mathbb{R}.$

- **Pointer**: This function controls the pointer-events on screen, where $x$ represents the coordinates for those events on the game space.

  $pointer(\ x\ ), x \in \mathbb{R}.$

- **Keyboard**: This function oversees the system's keyboard events, where $x$ is the code for a specific key and *label* represents the event mode.

  $keyboard(\ x, label\ ), x \in \mathbb{R}, label \in [press, release].$

- **Collide**: This function watches the collision detection between game actors, where $Ag$ is the actor to report a collision.

  $collide(\ Ag\ ), Ag \in AG.$

**Action Predicate *Do***

An action is defined as a specific behaviour to be performed by an actor. In the game engine, the actions are the non-logical function elements that represent the actions $\alpha \in Ac$ of the system. The actions can contain variables as arithmetic expressions in the same way as for the conditions.

This set of actions is based on the *Create*, *Read*, *Update* and *Delete* (CRUD) functions of information persistence on databases [36] applied to the actors. From these actions, the system can generate more complex actions that increase the abstraction level of the behaviour specification. With all this, the set of actions $Ac$ available in the system consists of the following actions $\alpha$:

- **Create**: It creates a new actor in the game as a copy of an existing actor $Ag$ in the environment.

$$\mathit{create}(\ Ag\ ),\ Ag \in AG.$$

- **Read**: It allows reading the information of a *property* that may belong to an actor $Ag \in AG$ or to the environment $Env$ of the game.

$$\mathit{read}(\ \mathit{property}\ )$$

- **Update**: It modifies the value of a *property* that can belong to an actor $Ag$ or the environment $Env$ of the game. The new value is determined from the evaluation of an arithmetic *expression.*

$$\mathit{update}(\ \mathit{property},\ \mathit{expression}\ )$$

- **Delete**: It removes an actor $Ag$ from the game environment.

$$\mathit{delete}(\ Ag\ ),\ Ag \in AG$$

## 5. Discussion

The proposed engine allows creating a wide variety of video games but also has some features that are interesting to analyse. In the first place, it has been tried to design a system that formally creates video games, based on the MAS analysis. Thereby, the engine has the essential features to create a large variety of games without other attributes from the conventional 2D game engines which, in experimentation have been proved as not necessary.

Next, it is necessary to emphasise that the games are made with a single type of actor and that there are no hierarchical relations between them. All the elements that define a game: the markers, the player, the NPC and so on, have the same structure of properties and behaviour rules. Also, the use of a scene graph is not necessary, which simplifies the engine internal architecture and the design of the game.

The properties defined for the game environment and the actors allow only data types such as text strings, numbers and booleans. There are no complex data structures such as vectors or matrices, which are not necessary for the creation of most arcade games. For instance, a *CandyCrush*-like game seems to require a matrix for its setup, but it can be created from a MAS perspective by using multiple agents arranged in rows and columns, including their suited properties and behaviour rules.

Besides that, if the behaviour specification language for the actors is analysed in detail, it can be verified that it is possible to define game behaviour using a set of predicate formulas of just five conditions and four actions, as it has been stated in the previous section. This is one of the main features of the proposed engine since it allows the creation of games without using complex scripting languages as in other game engines. Also, an analysis of this predicate language brings forward that logical operators are not used and it is only necessary to use the IF-THEN-ELSE structure in a nested way to define behaviour. Not even loops are required, since the game cycle itself performs a sequential evaluation of each of the actor's behaviour rules per cycle and it can, therefore, be avoided.

In the following section, it is presented a detailed description of some video games developed with the engine.

## 6. Use Cases

In order to test if the proposed game engine is capable of generating fully functional games that comply with the properties of a MAS, the game logic for three games has been constructed using the formulation presented in the previous sections. Specifically, the first of these games is the *Wolf-Sheep Predation*, which is based on a classic MAS problem; the other two are classic arcade games: *Frogger* and *Pac-Man*. In these games, the origin of the coordinate system is in the centre of the screen and it has positive and negative values.

Finally, an additional set of different arcade games have been created with the engine to demonstrate its potential.

### 6.1. Wolf-Sheep Predation

In this first game, the mechanics consist of a set of actors representing *Wolves* and *Sheeps* trying to survive in the game environment. These actors move arbitrarily through the stage, and every movement implies an expenditure of energy. Also, there is another actor representing the *Grass*, spread evenly on the stage. When a collision occurs between a *Sheep* actor and a *Grass* actor, the latter is destroyed, and the energy of the *Sheep* actor increases. Similarly, when a *Wolf* actor collides with a *Sheep* actor, it is also destroyed and the *Wolf* actor's energy increases. The reproduction of both actors occurs from time to time, depending on the game settings, as long as they have a sufficient level of energy.

Additionally, in the implementation of this game, there are three auxiliary actors in charge of generating the initial distribution of the three main agents. These auxiliary actors spawn new *Sheep*, *Wolf* and *Grass* actors initialised with an energy value and an initial arbitrary movement. Due to these three actors are not involved in the central dynamics of the game, they are not explained in the following game's algorithms.

**Sheep**: At first, the *Sheep* actors energy value is set at 300, and also their ρ1 rule gives them an initial arbitrary movement.

*initialization (ρ1)* ≡ { *If( compare( init*, TRUE, *equal ) )* → *Do( update( velocity*, *random(* -100, 100 ) ) ) ∧ *Do( update( init*, FALSE ) ) }

Besides that, when they collide with the limits of the screen, they change their direction, as it is indicated by their ρ2 and ρ3 rules. When colliding with a *Wolf* actor, the specific *Sheep* actor must be destroyed according to its ρ4 rule.

*wallXCollision (ρ2)* ≡ { *If( collide( wallX ) )* → *Do( update( velocity.x*, -1 * *velocity.x ) ) )* }

*wallYCollision (ρ3)* ≡ { *If( collide( wallY ) )* → *Do( update( velocity.y*, -1 * *velocity.y ) ) )* }

*wolfCollision (ρ4)* ≡ { *If( collide( wolf )* → *Do( delete( sheep ) )* }

In respect of the *Sheep* actor reproduction, the ρ5 rule controls the spawn of the actor and the energy loss. It is dependant on a random expression parametrized to trigger it on the 2% of the cases. If it happens, the *Sheep* actor would lose 50 energy units. In order to recover energy, it has to "eat" *Grass* actors to gain 50 energy units.

*reproduce (ρ5)* ≡ { *If( compare( random(* 0, 100 ), 2, *less ) )* → *If( compare( energy*, 50, *greaterEqual ) )* → *Do( create( sheep ) )* ∧ *Do( update( energy*, *energy* - 50 ) ) }

*grassCollision (ρ6)* ≡ { *If( collide( grass ) )* → *Do( update( energy*, *energy* + 50 ) ) }

This feeding process is essential, since each displacement implies an energy expenditure, specifically of an energy unit, as it can be seen in its ρ7 rule. By extension, if the energy reaches zero, the *Sheep* actor dies according to its ρ8 rule.

*walk (ρ7)* ≡ { *Do( update( energy*, *energy* - 1 ) ) }

| | | |
|---|---|---|
| *death* (*ρ8*) | ≡ | { *If*( *compare*( *energy*, 0, *lessEqual* ) ) → Do( *delete*( *sheep* ) ) } |

**Wolf**: Similarly to the *Sheep* actors, the *Wolf* actors have an initial energy value equal to 300 and their initial movement is determined by their ρ1 rule.

| | | |
|---|---|---|
| *initialization* (*ρ1*) | ≡ | { *If*( *compare*( *init*, TRUE, *equal* ) ) → Do( *update*( *velocity*, *random*( -100, 100 ) ) ) ∧ Do( *update*( *init*, FALSE ) ) } |

The rest of its rules are identical to the *Sheep* actor ones, but exchanging the *Grass* actors as feed for *Sheep* actors as it is indicated by their ρ4 rule.

| | | |
|---|---|---|
| *wallXCollision* (*ρ2*) | ≡ | { *If*( *collide*( *wallX* ) ) → Do( *update*( *velocity.x*, -1 * *velocity.x* ) ) } |
| *wallYCollision* (*ρ3*) | ≡ | { *If*( *collide*( *wallY* ) ) → Do( *update*( *velocity.y*, -1 * *velocity.y* ) ) } |
| *sheepCollision* (*ρ4*) | ≡ | { *If*( *collide*( *sheep* ) )→ Do( *update*( *energy*, *energy* + 50 ) ) } |
| *reproduce* (*ρ5*) | ≡ | { *If*( *compare*( *random*( 0, 100 ), 2, *less* ) ) → *If*( *compare*( *energy*, 50, *greaterEqual* ) ) → Do( *create*( *wolf* ) ) ∧ Do( *update*( *energy*, *energy* - 50 ) ) } |
| *walk* (*ρ6*) | ≡ | { Do( *update*( *energy*, *energy* - 1 ) ) } |
| *death* (*ρ7*) | ≡ | { *If*( *compare*( *energy*, 0, *lessEqual* ) ) → Do( *delete*( *wolf* ) ) } |

**Grass**: In this case, the purpose of this actor is to feed the *Sheep* actors. Thus, through its ρ1 rule, the *Grass* actors detect when a *Sheep* actor collides with them, and in that case, they proceed to destroy themselves.

| | | |
|---|---|---|
| *sheepCollision* (*ρ1*) | ≡ | { *If*( *collide*( *sheep* ) ) → Do( *delete*( *grass* ) ) } |

With the *Wolf-Sheep Predation* case, it has become clear that the proposed system is able to generate autonomous entities capable of generating complex behaviours from simple rules. It is important to highlight that the dynamics of the three principal actors of this game present similar behaviours, where the same event is treated separately by each involved agent, according to the role assigned to it in the predator-prey relationship. A screenshot of the game's run can be seen in Figure 2.

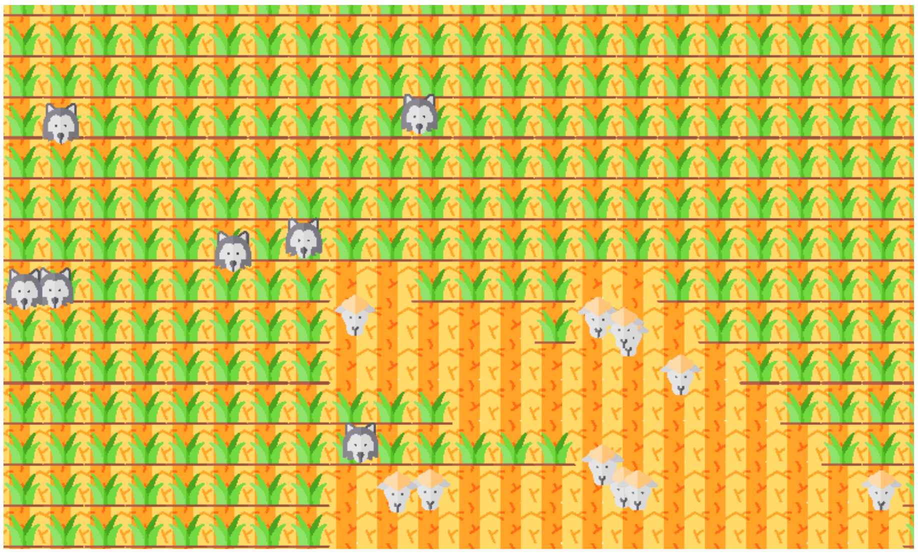

*Figure 2 .- A capture of the Wolf-Sheep Predation game.*

### 6.2. Frogger

The *Frogger* game consists of a *Frog* actor trying to cross a road with a series of *Car* actors passing by and a river with *Trunk* actors that must be used to reach the other side. The *Car* actors and the *Trunk* actors are initialised with a constant speed that makes them move from right to left. Also, there is an actor representing the *Water* area and two other auxiliary actors working as *Car* and *Trunk* generators. As it is shown in Figure 3, there are five lanes through which *Car* and *Trunks* actors circulate. Each one of these lines has an auxiliary actor that spawns *Car* or *Trunk* actors based on a timer function, which is controlled by a random expression to make its appearance in the game less predictable.

**Frog**: The *Frog* can move in any direction that the user determines with its ρ1, ρ2, ρ3 and ρ4 rules.

| | | |
|---|---|---|
| *leftKey* (*ρ*1) | ≡ | { If( *keyboard( left, press)* ) → Do( *update( position.x, position.x* - 10 ) ) ∧ Do( *update( angle*, 270 ) ) } |
| *rightKey* (*ρ*2) | ≡ | { If( *keyboard( right, press)* ) → Do( *update( position.x, position.x* + 10 ) ) ∧ Do( *update( angle*, 90 ) ) } |
| *upKey* (*ρ*3) | ≡ | { If( *keyboard( up, press)* ) → Do( *update( position.y, position.y* + 10 ) ) ∧ Do( *update( angle*, 180 ) ) } |
| *downKey* (*ρ*4) | ≡ | { If( *keyboard( down, press)* ) → Do( *update( position.y, position.y* - 10 ) ) ∧ Do( *update( angle*, 0 ) ) } |

If it collides with a *Car* actor, it is removed from the game as it is set up in its ρ5 rule.

| | | |
|---|---|---|
| *destroyByCar* (*ρ*5) | ≡ | { If( *collide( car )* ) → Do( *delete( frog )* ) } |

In the same way, with its ρ6 rule, if it collides with the *Water* actor and is not in contact with a *Trunk* actor, it is also deleted from the stage.

*destroyByDrowning* (*ρ*6) ≡ { If( *collide*( *water* ) ) → ¬If( *collide*( *trunk* ) ) → Do( *delete*( *frog* ) ) }

When it lands on a *Trunk* actor, as it is indicated by its ρ7 rule, it inherits the movement from right to left from the *Trunk* actor.

*trunkCollision* (*ρ*7) ≡ { If( *collide*( *trunk* ) ) → Do( *update*( *velocity.x*, -100 ) ) }

**Car**: *Cars* always move from right to left. If they reach the left edge of the screen, they are eliminated from the game as indicated by their ρ1 rule.

*destroy* (*ρ*1) ≡ { If( *compare*( *position.x*, -*screen.width* / 2 - *width*, *lessEqual* ) ) → Do( *delete*( *car* ) ) }

**Trunk**: As in the previous actor, it moves from right to left until it is entirely out of the screen as it can be seen at its ρ1 rule.

*destroy* (*ρ*1) ≡ { If( *compare*( *position.x*, -*screen.width* / 2 - *width*, *lessEqual* ) ) → Do( *delete*( *trunk* ) ) }

With the *Frogger* game, a classic arcade game has been generated where all entities have completely autonomous behaviour and whose orchestration generates an elaborate game. Simple actor dynamics determine the mechanics of this game based on simple movements and interactions between the *Frog* actor and the other actors present in the game. In Figure 3, a capture of the *Frogger* game during its run on the game engine is shown.

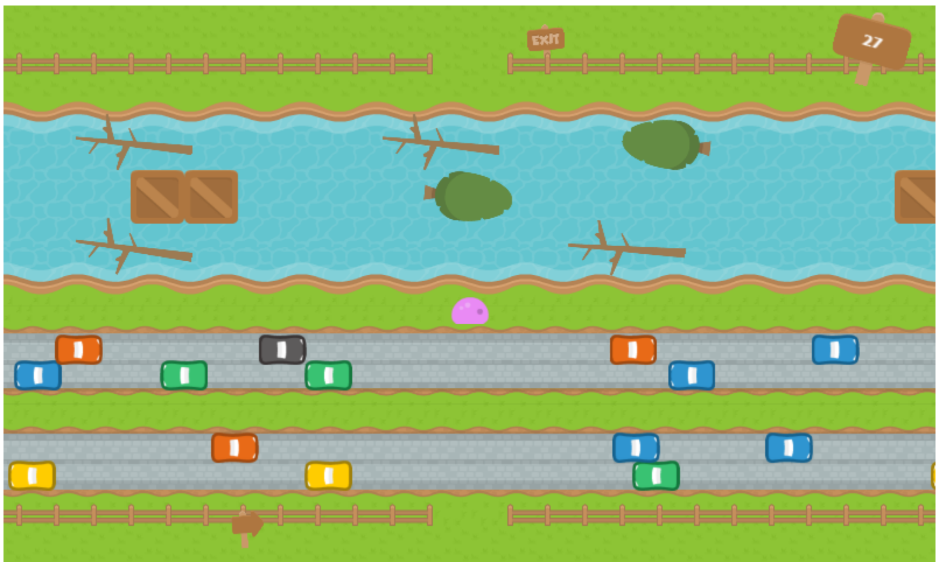

*Figure 3 .- A capture of the Frogger game.*

### 6.3. Pac-Man

The *Pac-Man* actor is in a maze along with four *Ghost* actors. These actors can move in any direction as long as they do not collide with the maze walls. If the *Pac-Man* actor collides with a *Ghost* actor, it loses the game. Throughout the alleys of the maze, a set of *Food* actors are arranged to provide points to the *Pac-Man* actor after a collision with them. Also, there is a set of *Coin* actors to allow the *Pac-Man* actor to chase and eliminate the *Ghost* actors. When the *Pac-Man* actor collides with a *Coin* actor, a short time begins in which it can chase the *Ghost* actors and destroy them.

**Pac-Man**: The *Pac-Man* actor is initialised with zero points and with a constant speed. The user controls the direction of its movement through its ρ1, ρ2, ρ3 and ρ4 rules.

| | | |
|---|---|---|
| *leftKey (ρ1)* | ≡ | *{ If( keyboard( left, press ) ) → Do( update( velocity.x, -100 ) ) ∧ Do( update( velocity.y, 0 ) ) ∧ Do( update( angle, 180 ) ) }* |
| *rightKey (ρ2)* | ≡ | *{ If( keyboard( right, press ) ) → Do( update( velocity.x, 100 ) ) ∧ Do( update( velocity.y, 0 ) ) ∧ Do( update( angle, 0 ) ) }* |
| *upKey (ρ3)* | ≡ | *{ If( keyboard( up, press ) ) → Do( update( velocity.y, 100 ) ) ∧ Do( update( velocity.x, 0 ) ) ∧ Do( update( angle, 90 ) ) }* |
| *downKey (ρ4)* | ≡ | *{ If( keyboard( down, press ) ) → Do( update( velocity.y, -100 ) ) ∧ Do( update( velocity.x, 0 ) ) ∧ Do( update( angle, 270 ) ) }* |

If it collides with the maze walls, it stops as indicated by its ρ5 rule.

| | | |
|---|---|---|
| *wallCollision (ρ5)* | ≡ | *{ If( collide( wall ) ) → Do( update( velocity, 0 ) ) }* |

Besides that, if it collides with a *Food* actor, its points property increases as it is marked by its ρ6 rule. However, if it collides with a *Coin* actor, as its ρ7 rule indicates, the chase mode is activated during the time determined by its ρ8 rule.

| | | |
|---|---|---|
| *food (ρ6)* | ≡ | *{ If( collide( food ) ) → Do( update( points, points + 1 ) ) }* |
| *coins (ρ7)* | ≡ | *{ If( collide( coin ) ) → Do( update( points, points + 10 ) ) ∧ Do( update( chase, TRUE ) ) }* |
| *chase (ρ8)* | ≡ | *{ If( compare( chase, TRUE, equal ) ) → If( timer( 5 ) ) → Do( update( chase, FALSE ) ) }* |

During that time frame, the *Pac-Man* actor can eliminate *Ghost* actors as it is determined at ρ9 rule. Conversely, if it collides with a *Ghost* actor and the chase mode is not active, the *Pac-Man* actor is destroyed.

| | | |
|---|---|---|
| *enemyCollision (ρ9)* | ≡ | *{ If( collide( ghost ) ) → If( compare( chase, TRUE, equal ) ) → Do( update( points, points + 50 ) ) ∧ ¬If( compare( chase, TRUE, equal ) ) → Do( delete( pacman ) ) }* |

**Ghost**: The *Ghost* actors behaviour is initialised by a constant speed and by their ρ1 rule, which controls the direction of the actor after colliding with the maze walls. It is necessary to point out that the random function is actually just run once, but it is displayed twice to fulfil the positive and negative ways of the predicate If on the IF-THEN-ELSE rule structure.

| | | |
|---|---|---|
| *wallCollision (ρ1)* | ≡ | *{ If( collide( wall ) ) → If( compare( random( -1, 1 ), 0, less ) ) → Do( update( velocity.x, velocity.x * -1 ) ) ∧ ¬If( compare( random( -1, 1 ), 0, less ) ) → Do( update( velocity.x, velocity.x * -1 ) ) }* |

Also, they have the ρ2 rule that controls if the *Pac-Man* actor is in chase mode; in that case, they must change their display image to a "scared" one.

| | | |
|---|---|---|
| *scared (ρ2)* | ≡ | *{ If( compare( chase, TRUE, equal ) ) → Do( update( image, scaredImage ) ) ∧ ¬If( compare( chase, TRUE, equal ) ) → Do( update( image, normalImage ) ) }* |

Similarly, in this case, if they collide with the *Pac-Man* actor, they are removed as it is indicated by their ρ3 rule.

| | | |
|---|---|---|
| *destroy (ρ3)* | ≡ | *{ If( collide( pacman ) ) → If( compare( chase, TRUE, equal ) ) → Do( delete( ghost ) ) }* |

The *Pac-Man* game has complex features that could be solved with this approach based on MAS. Both the *Pac-Man* actor and the *Ghost* actors have similar behaviours, the difference lies in the decision-making process for the direction change: the *Pac-Man* actor relies on the interaction with the game user while the *Ghost* actors decide to change direction arbitrarily after colliding with the maze walls. Simultaneously, the *Ghost* actors change their behaviour when the *Pac-Man* actor is in chase mode. This behaviour is derived from the perception of the *Ghosts* actors on the state of the *Pac-Man* actor. A capture of the game during its run is shown in Figure 4.

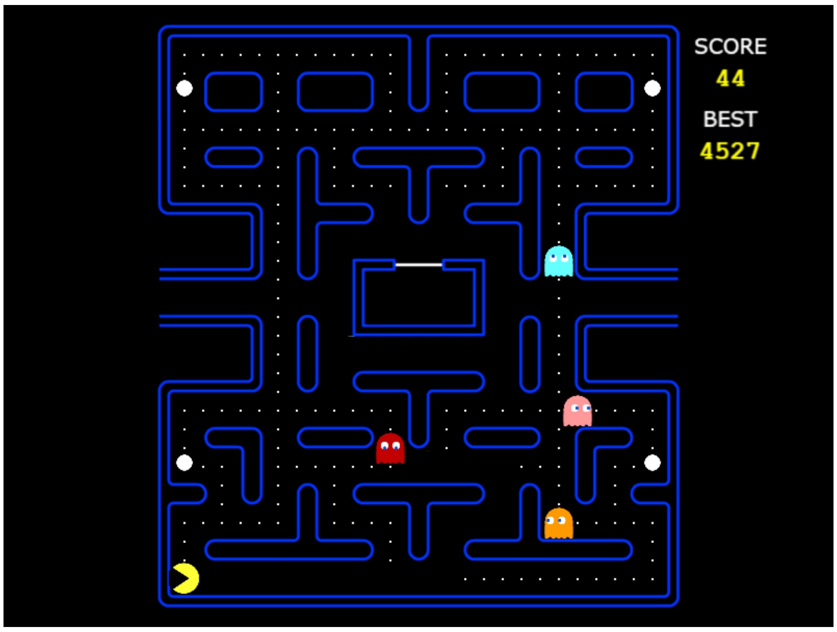

*Figure 4 .- A capture of the Pac-Man game.*

### 6.4. Other Games Developed

Independently from the use cases analyzed in this section, a series of additional games have also been developed to support the validation of the system's competences. These games tried to test multiple video game genres and their respective game mechanics. The games under consideration are the following:

- **Arkanoid**. A classic puzzle game where the ball bounces until there are no bricks left.
- **Asteroids.** A classic space and third-person shooter where the asteroids are subdivided after being hit.
- **Groñof! Adventures**. A pointer-based game in which the player must eliminate enemies to move to the next level.
- **Ducks**. A first-person shooter game based on the classic ones found in fairgrounds.
- **Diamond Crush.** A match-three puzzle video game, in which the players have to swap diamonds on a game board to produce combinations of three or more with the same colour.
- **Fallas the Game**. A physics-based game, inspired by the well-known Angry Birds game, where the player has to burn the target to ashes.
- **Abstract Adventure**. A classic platform game where the goal is to elude enemies and traps at the same time as every coin in the scene is gathered.
- **Cowboys vs Aliens**. Tower defence game where the Cowboys defend their land from the Aliens.
- **Afterglow**. A horizontal scroll game, where the goal is to travel for as long as possible shooting and avoiding collisions with the enemies and their shoots.

In Figure 5, captures of these nine games during their run are presented in the same order of the previous list. Additionally, a set of these games presented in this section have been arranged on a web page [37].

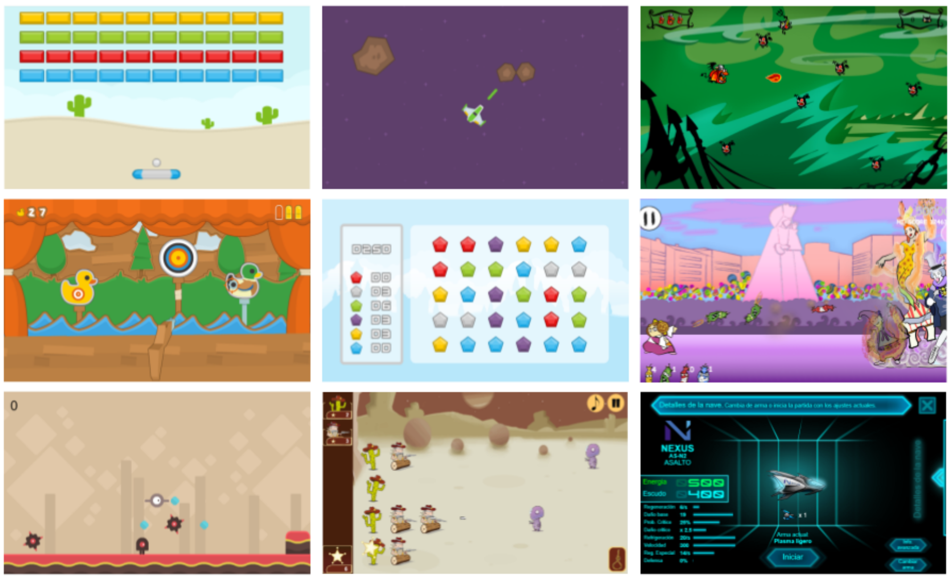
Figure 5 .- Captures of the games developed in the game engine.

## 7. Conclusions and Future Work

This work presents the definition of a game engine able to produce games that meet the requirements of a MAS. It draws from the formal definition of the MAS to conduct a description of the game and its essential elements. The game represents the environment and the actors are the agents of the multi-agent games generated by this game engine. From the environment state, the actors can perceive information and react to specific states based on their predefined tasks. The behaviour associated with the tasks is determined by sets of behaviour rules and a pre-established logic semantics.

The construction of the game is based on a unique type of actor and without hierarchical relations between them. Each one of them has the same structure of properties and behaviour rules. All this without a scene graph, which simplifies the engine internal architecture. Moreover, the specification of the engine has reached a level of abstraction for the actor's behaviour definition that makes unnecessary complex data structures such as vectors or matrices during the creation of most arcade games.

Regarding the actor's behaviour specification, it has been proved that just five conditions and four actions are enough to define games using predicate formulas. It avoids the scripting for the creation of games, making that as one of the main features of the proposed engine. Also, all this is achieved without logical operators, matrices and loops, since the game cycle performs cyclical evaluations of the behaviour rules, just by using the IF-THEN-ELSE structure.

The use cases presented in this article show the potential of the game engine for the generation of games as MAS. Besides that, it has been observed that the system can generate logic, mechanics and complex behaviours from a very reduced first-order logic semantics as a behaviour definition descriptors.

As future work, the possibility of applying this methodology to the development of game engine architecture is proposed, in order to formalise an underdeveloped research field. Thus, the game engines design could be reformulated and the parallelisation of their behaviour could be explored.

## 8. Acknowledgements

This work has been supported by the Ministry of Science and Technology (TIN2016-75866-C3-1-R and RTI2018-098651-B-C54) and the Universitat Jaume I research projects (UJI-B2018-56 and UJI-B2018-44). Furthermore, this work has been made possible thanks to the graphics resources created by Kenney from Kenney.nl.

## 9. References

1. Gregory, J. Game engine architecture. 2017. AK Peters/CRC Press.
2. Nystrom, R. Game programming patterns. 2014. Genever Benning.
3. Anderson, E. F., Engel, S., McLoughlin, L., & Comninos, P. The case for research in game engine architecture, 2008; 228-231.
4. Ampatzoglou, A., & Stamelos, I. Software engineering research for computer games: A systematic review. Information and Software Technology, 2010; 52(9), 888-901.
5. Dorri, A., Kanhere, S. S., & Jurdak, R. Multi-agent systems: A survey. IEEE Access, 2018; 6, 28573-28593.
6. Silva, C. T., Castro, J., & Tedesco, P. A. Requirements for Multi-Agent Systems. WER, 2003; 198-212.
7. Biswas, P. K. Towards an agent-oriented approach to conceptualization. Applied Soft Computing, 2008; 8(1), 127-139.
8. Grant, M., Sandeep, V., & Fuhua, L Integrating Multiagent Systems into Virtual Worlds. In 3rd International Conference on Multimedia Technology, 2013; 581-588.
9. Olfati-saber, R. Flocking For Multi-agent Dynamic Systems: Algorithms And Theory. Technical Report CIT-CDS 2004-005, California Inst Of Tech Pasadena Control And Dynamical Systems, 2004.
10. Olfati-Saber, R., Fax, J. A., & Murray, R. M. Consensus and cooperation in networked multi-agent systems. Proceedings of the IEEE, 2007; 95(1), 215-233.
11. Wooldridge, M. An introduction to multiagent systems. 2009. John Wiley & Sons.
12. Doherty, M. A software architecture for games. University of the Pacific Department of Computer Science Research and Project Journal, 2003; 1(1).

13. Lewis, M., & Jacobson, J. Game engines. Communications of the ACM, 2002; 45(1), 27-31.
14. Millington, I., & Funge, J. Artificial intelligence for games. 2009. CRC Press.
15. Dignum, F., Westra, J., van Doesburg, W. A., & Harbers, M. Games and agents: Designing intelligent gameplay. International Journal of Computer Games Technology, 2009. ID-837095.
16. Jepp, P., Fradinho, M., & Pereira, J. M. An agent framework for a modular serious game. In 2nd International Conference on Games and Virtual Worlds for Serious Applications, 2010; 19-26.
17. Aranda, G., Carrascosa, C., & Botti, V. Characterizing massively multiplayer online games as multi-agent systems. In International Workshop on Hybrid Artificial Intelligence Systems, 2008; 507-514.
18. Aranda, G., Trescak, T., Esteva, M., Rodriguez, I., & Carrascosa, C. Massively multiplayer online games developed with agents. In Transactions on Edutainment VII, 2012; 129-138.
19. Garcés, A., Quirós, R., Chover, M., & Camahort, E. Implementing moderately open agent-based systems. In IADIS International Conference WWW/Internet, 2006; 360-369.
20. Garcés, A., Quirós, R., Chover, M., Huerta, J., & Camahort, E. A development methodology for moderately open multi-agent systems. In Proceedings of the 25th conference on IASTED International Multi-Conference: Software Engineering, 2007; 7, 37-42.
21. Garcés, A., Quirós, R., Chover, M., & Camahort, E. Implementing virtual agents: a HABA-based approach. Int J Multimed Appl, 2010; 2, 1-15.
22. Remolar, I., Garcés, A., Rebollo, C., Chover, M., Quirós, R., & Gumbau, J. Developing a virtual trade fair using an agent-oriented approach. Multimedia Tools and Applications, 2015; 74(13), 4561-4582.
23. Ramos, F., Chover, M., Ripolles, O., & Granell, C. Continuous level of detail on graphics hardware. In International Conference on Discrete Geometry for Computer Imagery, 2006. 460-469.
24. Ripolles, O., Ramos, F., Puig-Centelles, A., & Chover, M. Real-time tessellation of terrain on graphics hardware. Computers & geosciences, 2012; 41, 147-155.
25. Sacerdotianu, G., Ilie, S., & Badica, C. Software Framework for Agent-Based Games and Simulations. In 13th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, 2001; 381-388.
26. Adobbati, R., Marshall, A. N., Scholer, A., Tejada, S., Kaminka, G. A., Schaffer, S., & Sollitto, C. 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, 2001; 5, 6.
27. Pons, L., & Bernon, C. A Multi-Agent System for Autonomous Control of Game Parameters. In IEEE International Conference on Systems, Man, and Cybernetics, 2013; 583-588.

28. Unity 3D Engine. Unity Technologies, 2019. http://www.unity3d.com [Online; Last accessed: 2019-4-15] (April 2019).
29. Becker-Asano, C., Ruzzoli, F., Hölscher, C., & Nebel, B. A Multi-Agent System based on Unity 4 for virtual perception and wayfinding. Transportation Research Procedia, 2014; 2, 452-455.
30. Finnsson, H., & Björnsson, Y. Simulation-Based Approach to General Game Playing. In AAAI, 2008; 8, 259-264.
31. Fagin, R., Halpern, J. Y., Moses, Y., & Vardi, M. Y. Reasoning about knowledge. 1995. MIT Press.
32. Russell, S. J., & Subramanian, D. Probably bounded-optimal agents. Journal of Artificial Intelligence Research, 1994; 2, 575-609.
33. Genesereth, M. R., & Nilsson, N. J. Logical foundations of artificial. Intelligence. 1987. Morgan Kaufmann; Chapter 6.
34. Brachman, R. J., Levesque, H. J., & Reiter, R. Knowledge representation. 1992. MIT press.
35. Karplus, K. Using if-then-else DAGs for multi-level logic minimization. Technical Report UCSC-CRL-88-29, Computer Research Laboratory, University of California, 1988.
36. Daissaoui, A. Applying the MDA approach for the automatic generation of an MVC2 web application. In 4th International Conference on Research Challenges in Information Science, 2010; 681-688.
37. Games developed as demonstrators in the use cases. Retrieved July 30, 2019, from https://sites.google.com/uji.es/multiagent-gameengine