Hindawi Publishing Corporation
International Journal of Computer Games Technology
Volume 2009, Article ID 837095, 18 pages
doi:10.1155/2009/837095

Research Article

Games and Agents: Designing Intelligent Gameplay

F. Dignum,[1] J. Westra,[1] W. A. van Doesburg,[2] and M. Harbers[1, 2]

[1] *Department of Information and Computing Sciences, Utrecht University, P.O. Box 80.089, 3508 TB Utrecht, The Netherlands*
[2] *TNO Defence, Security and Safety, P.O. Box 23, 3769 ZG Soesterberg, The Netherlands*

Correspondence should be addressed to F. Dignum, dignum@cs.uu.nl

Received 27 April 2008; Revised 20 August 2008; Accepted 2 December 2008

Recommended by Abdennour El Rhalibi

There is an attention shift within the gaming industry toward more natural (long-term) behavior of nonplaying characters (NPCs). Multiagent system research offers a promising technology to implement cognitive intelligent NPCs. However, the technologies used in game engines and multiagent platforms are not readily compatible due to some inherent differences of concerns. Where game engines focus on real-time aspects and thus propagate efficiency and central control, multiagent platforms assume autonomy of the agents. Increased autonomy and intelligence may offer benefits for a more compelling gameplay and may even be necessary for serious games. However, it raises problems when current game design techniques are used to incorporate state-of-the-art multiagent system technology. In this paper, we will focus on three specific problem areas that arise from this difference of view: synchronization, information representation, and communication. We argue that the current attempts for integration still fall short on some of these aspects. We show that to fully integrate intelligent agents in games, one should not only use a technical solution, but also a design methodology that is amenable to agents. The game design should be adjusted to incorporate the possibilities of agents early on in the process.

Copyright © 2009 F. Dignum et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

1. Introduction

The gaming industry has changed dramatically over the past few years. Where the development focus used to be on the graphical possibilities of the games, that is, the naturalness of the image rendering, the near movie realism of the graphics now increasingly contrasts with the rather primitive and unnatural behavior of the characters. With behavior, we do not mean the animation of the character, but its cognitive behavior, that is, the reaction and interaction with other characters, the consistency over time of its goals, and so forth. Therefore, the focus is now shifting toward more natural behavior of the game characters. This shift changes the focus on the techniques to be used as well. Whereas geometric techniques and graphics were the prime focus, now it seems the time to introduce more serious AI techniques, see for example [1]. We see an increasing use of techniques such as fuzzy logic and neural networks to enhance the decision functions of the characters, see [2] for an example. These features are very useful to make individual behaviors look more realistic and in some cases make them blend into a crowd in a natural way.

The game developers community has also recognized the importance of making characters appear intelligent over longer periods of time. Finite state machines are most often used to model the life cycle behavior of a character. Each state describes an important state of the character that determines its choice of available actions. Although this works fine for simple behavior, Orkin [3] realized that more flexible planning is needed for complex behavior. In F.E.A.R., planning techniques based on STRIPS [4] and goal-oriented action planning [5] are introduced. This leads to more natural behavior because the goals of the character are separated from the plans generated to reach the goal. Therefore, the failure of a plan does not directly lead to giving up a goal but rather leads to generating an alternative plan including the new information about the world that led to the failure of the first plan. Interestingly enough though, little reference is made in this work to the research performed in the agent community about dynamic and real-time planning, which would be directly applicable such as [6].

Despite these examples, few commercial games have focused on making characters within games behave more natural on a *cognitive* level. Probably the main exception was the Soar Quakebot build in SOAR for Quake II. However, this was done (successfully) as an academic project, but the Soar Quakebot was not incorporated in later commercial versions of Quake. This is partly explained by the fact that Quake like most video games does not require more complicated behavior than that of film characters played by actors like Sylvester Stalone or Arnold Schwarzenegger. Not much intelligence is needed to emulate that behavior yet. However, if we want to move beyond the shooting and fighting games toward games in which multiple characters interact naturally over extended episodes or serious games to train people leading teams in stressful situations, we need more cognitive believable behavior of the characters. One of the problems mentioned in [3] is the implementation of believable and natural communication between characters. This can be done by giving them additional information, not available to normal players. Also the use of special environmental characteristics can provide the illusion that characters are cooperating, while they merely all react to the same environmental cue. These features are for instance, used in F.E.A.R. to create a realistic appearance. However, this trick can only be used in well-defined environments and everything has to be preprogrammed.

The more complex the games become and the more elaborate the interactions between the characters during the game, the more difficult it will become to design these characters without the use of specialized tools geared toward implementing intelligent agents in a modular way. We aim for characters that are programed using agent technology that actually incorporates deliberation on actions and cooperation, rather than simulated intelligence through clever tricks. Thus this seems an excellent area for applying intelligent agent technology such as that being developed within computer science faculties at the universities, which already for a decade has developed models, techniques, and tools to design software based on design concepts such as goals, intentions, plans, and beliefs. Some first attempts to connect game engines with this type of agents have been made, for example [7, 8]. However, these are very specific to game engine and agent type, no general solutions have been proposed. The main issue of this paper is thus, given that it makes sense to use ideas from agent research in gaming, as seems to be supported by the growing amount of work in games that incorporates (parts of) agent concepts and technologies, what would be necessary to make use of the agent technology as developed for the multiagent platforms?

The techniques used in agent technology do not seamlessly fit with those used in game technology. Agent technology has hardly bothered about efficiency issues up till now. Most applications are not real-time ones or still have large time scales. Moreover, agent technology usually assumes distributed control and a certain level of autonomy of the agents. This is in stark contrast with game technology in which the game engine dictates the application, and strict time constraints are used in order to render the images naturally and efficiently.

In this paper, we explore the possibilities to join the game to existing agent technologies despite some inherent incompatibilities. We will focus on some of the most apparent problem areas and also show how they can be solved in a structural way. A main assumption of this paper is that we want to use agent platforms and the associated technology to develop the intelligent agents, because platforms such as JADEX, Jack, Jason, and 2APL [9] provide optimal support for developing the agents themselves. This will, therefore, support the design principle of separation of concerns, which is important for complex systems. The way to design a complex system is to separate different concerns and tackle them separately using the most appropriate tools for that part and joining them later. This principle can already be seen in the game technology where the game engine is built from physics engine, animation engine, and so forth, each taking care of one part of the design of the game. So, the challenge is to connect the agents developed on these platforms to the game engines on which the rest of the game is developed. The work reported in this paper is based on our experiences of connecting the 2APL platform to several game engines. These experiences led us to the conviction that in order to fully integrate agents in games, one should not only use a technical solution, but also a design methodology that is amenable to agents. We aim to support this claim as well in this paper. The areas that we will specifically look at are: synchronization, information representation, and communication.

As the agents will run in separate threads from the game engine (in principle using the agent platform), the actions of the agents and the game engine also have to be synchronized explicitly, for example, in order to make the agents behave according to the laws of physics. The problem of synchronization is of course not new; neither will we present completely new solutions. However, the solution should take the peculiarities of the games and agents into consideration.

Information filtering is needed to provide both the agents and the game engine with the right type of information at the right time. Whereas the geometric information might be the most important for the game engine, the agent wants to get its information at a higher knowledge level. For example, instead of knowing the rotation angle of a rectangular object, the agent just wants to know that the door is open. Conversely, the agent might perform an action to move a fire hose toward the house which has to be translated to more geometric actions that can be used by the game engine. The idea of using different knowledge levels to solve different types of problems dates back to Allen Newell [10] who distinguished, for example, a biological, cognitive, rational, and social level. Each knowledge level represents the information in a format that is suited for that particular type knowledge and may also contain its own type of problem solving methods. We actually propagate the same idea and claim that agents need a different (cognitive or rational) knowledge level from the game engine, which uses a biological (or physical) knowledge level.

Communication is the last area that we will consider explicitly. We will mainly look at communication between different characters in a game. In most games, coordination between characters is preprogrammed and thus communication is only needed on a small scale. In multiagent systems, communication is one of the pillars of the whole system and thus takes a prominent place in both design as well as technology. We will show how communication can be adequately integrated with the game engine to provide the agents with easy communication, while keeping it visible for the game engine.

The rest of the paper is ordered as follows. First, we will discuss the state of the art with respect to games and agents and identify problem areas. In Section 3, we describe some applications of (serious) games that rely on intelligent behavior and are currently difficult to achieve, but will be more readily attainable using our approach. In Section 4, we will describe our vision on connecting games and agents, using three different perspectives to alleviate the problems of Section 2 and enable games described in Section 3. In Section 5, we will discuss the different parts of our approach and indicate their contribution to the type of gaming scenarios described in Section 3. Finally, we will draw some conclusions and sketch directions for future work in Section 6.

2. State of the Art

In this section, we discuss several approaches to the integration of agents in game engines. As said in the introduction, many games already advertise the use of AI; however, the meaning of the term agent differs between game developers and AI researchers. In the next subsection, we discuss the type of agent we will focus on in this paper. Subsequently, we discuss different ways in which agents are connected to game engines.

2.1. Software Agents. In the game developer community, the term software agent usually refers to some character or unit within the game. In contrast, in the area of agent research, many definitions of software agents are used, which can lead to confusion when the term is used by different communities. (For some attempts to get to a common definition and characterize agents, see, e.g., [11]). However, there are some features of agents that are generally accepted in the community, which we will also adhere to in this paper. Software agents should be autonomous, proactive, reactive, and socially able. In this paper, we assume the following, more specific, definition of software agent: a software agent is a piece of software that has its own goals available (is proactive) and will try to achieve them without intervention of a user or other program (is autonomous), while sensing its environment and reacting to possible changes (is reactive). Additionally, agents in a multiagent system (MAS) are not centrally controlled, execute asynchronous, and should be able to communicate with each other, the user(s) and the environment. Software agents might be able to learn and adapt, but we do not consider these features as essential for agents.

The above definition mentions the generally accepted features of agents, but of course is still quite vague. However, it does give an indication of what type of features one generally expects in agents. Without going into a complete classification of agents, we do want to mention a few types of software agents that are already used in the context of gaming. Most important are the virtual agents or believable characters. They are especially useful for user interfaces and as such the emphasis is on natural interaction of the characters with persons, see [12, 13] for examples of these types of agents. The goals of these types of agents exist only implicit in the way that the rules with which they react to the environment are modeled and ordered in a way to resemble the fact that they have a goal. Because these types of agents usually have only one goal (something like assisting the user to understand or use the system), this will work fine. However, this approach fails when the character has more complicated goals or several goals that are competing. This happens particularly when more than one virtual agent is present at the same time and they have to cooperate which is usually avoided in this type of applications.

Much work on using multiple agents with (relatively) simple behavior is done in the area of (agent-based) social simulation (see, e.g., [14]). These agent systems focus on the emergent behavior of the system as a result of the interactions of the agents according to simple rules. Some work where multiple agents have been used in a simulation environment for training is [15]. In this work, the agents represent individuals or groups that interact in a virtual village, region, or country. Their goal is to study how the behavior of groups influences that of other groups or individuals. In these simulations, the agents are not really autonomous. They react to their environment through relatively simple rules. More importantly they do not plan but only execute actions one by one. Planning can be simulated by manipulating the environment in such a way that sequences of actions are forced after the first action is taken. However, it is difficult to program long-term goals in these agents.

In the research on multiagent systems (MASs), the starting point is that each agent represents a point of view or party with its own goal. Therefore, usually each agent runs in its own thread such that it can be autonomous. Also, agents usually contain some mechanism to deliberate about which action to take next in order to reach their own goal. The interactions between the agents emerge from the fact that the goals of the agents are not independent and thus the agents need each other to achieve their goals. Therefore, the design of agent interactions in such a way that all agents can reach their goal is of prime importance. This is illustrated by the amount of game theory-related research reported at the recent MAS conferences [16]. In MAS, the communication facilities play a crucial role. Because not all interactions are preprogrammed, a high degree of flexibility is needed to handle the communication. The de facto standard communication language is the FIPA ACL [17], which is based on speech act theory and can be used to pass information, but also to request or order actions. MAS platforms support communication by providing addresses of all agents, delivering messages in the right order, and so forth. The most widespread MAS platform is JADE(X), which is provided as a library of JAVA classes and fully supports the use of FIPA ACL communication (and thus provides easy interoperability with other FIPA ACL compliant platforms).

Good examples of applications of MAS are logistics and virtual organizations. In these business applications, the benefits of representing the stakeholders by their own agent that pursues the goal of that party while interacting with the other parties (either cooperatively or competitively) becomes obvious. It is this kind of MAS that we are aiming to connect to the games. Taking this type of MAS as a starting point provides a means to design each virtual character with its own goal, while being able to interact with other characters to reach its goal.

Also in MAS, there are several types of agents. We are particularly interested in *intelligent* agents, which will use some form of logic to perform their deliberation. That is, they are able to reason about their own goals and plans, to check which plan is best to achieve their goal given the current situation of the world, and to replan when the situation changed. The most well-known type of intelligent agents is the so-called BDI agent [18], which are specified (and sometimes implemented [19, 20]) in terms of the agent's beliefs, desires, and intentions. We believe that the BDI agents are most suitable to implement consistent long-term intelligent behavior in games. They seem a natural extension to the work started by the use of goal-oriented action planning in gaming as they also make explicit use of goals and planning. However, they also incorporate mechanisms to effectively use communication and other interaction mechanisms in their action deliberation.

Some platforms that are more geared toward the use of agents for cognitive simulation (and thus, like the BDI agents, seem suitable for use within the gaming area) are SOAR [21] and ACT-R [22]. In this paper, we do not commit to a particular platform, but rather try to propose a more generic framework that can be used by most agent types and platforms. We thus will only refer to properties that are shared by most (well-known) agent platforms.

2.2. Connecting Games and Agents. Current work on combining agent systems such as the described above and game environments either uses a server or client-side approach. The server-side approach can be said to be the traditional approach used in game design. In server-side approaches, the decision-making process of the agents is usually completely integrated into the game, resulting in agents that have to make decisions within one time step of the game loop. As such, server-side approaches have not made use of the available agent systems. Examples of this approach are Quake III [23], Never Winter Nights [24], F.E.A.R. [25], and Bos Wars [12]. In contrast, agents in client-side approaches are separate applications using the network information that is usually sent to a client game (a game instance that connects to a server for the world information, such as the one used by the human player). Some examples of this approach are Gamebot [26] connecting to Unreal Tournament 2003, Flexbot [27] connecting agents to Half-life, and Quakebots [28] in Quake II. Most of these types of implementations are made for research purposes.

Intelligent human-like behavior is important in the first person shooter games because in the newest ones, the agents control single avatars just like humans can control their avatar. Thus bots should be intelligent enough to perform in a way that makes their avatar resemble an avatar of a human player. In real-time strategy games, a whole team is controlled by a single agent. The team members are just executing basic instructions received from this top level control. In this setting the emphasis is more on the quality of the strategy and winning the game than on whether the strategy resembles that of human players. Finally, the pace of first person shooter games is higher than in most other game genres, necessitating quicker decision making and use of heuristics. For some games, it is claimed that more sophisticated agent technologies are used, but this is difficult to verify because most games are not open source. Most games also have no publications from the creators and third party publications are often inconsistent. So, in this paper, we use a rather old game (Quake III), but we believe very prototypical for this approach, to illustrate our point, because it was the only open source game that we could access and thus reliably discuss.

In order to get a better understanding of the state-of-the-art approaches, we will analyze Quake III as a prototypical example of the server-side approach and one of the few of which the code is open source and thus inspectable and Gamebots as a good example of the client-side approach. For both approaches, the analysis will be made according to three major aspects: synchronization, information representation, and communication.

2.2.1. A Server-Side Approach: Quake III. In Quake III, the agents are completely integrated in the default game loop in the same way as the physics engine, the animation engine, and rendering engine. The agent's decisions are defined by a sequence of method calls, and the methods return the action that has to be performed at that time step. Direct method calls can be used for many different decision-making processes, for example, hard coding approaches, directly specifying what to return with a certain input; fuzzy logic, mapping the right output to a certain set of input variables; or finite state machines, identifying the situation the agent is in and executing the corresponding method call. Independent of the particular decision-making strategy, the whole process is completely synchronized. This limits the complexity of behavior because in a synchronized process all decisions have to be made within one time step, and complex decisions would slow down a game too much.

Figure 1 gives an impression of the implementation of agents in Quake III. On the lowest level in the figure, a translation from the raw engine data to a representation more suitable for agents has been created, called the area awareness system (AAS). The heart of the AAS is a special 3D representation of the game world that provides all the information relevant to the bot. The AAS informs the bot about the current state of the world, including information relevant to navigation, routing, and other entities in the game. The information is formatted and preprocessed for fast and easy access and usage by the bot. For instance, to navigate the bot receives information from the AAS about the locations of all static items, and it can ask the AAS whether a certain location is reachable. The AAS is responsible for route planning. The first level also executes the actual actions of the agent and facilitates the decision process of the agents. However, the agents are highly dependent on the data they can extract from the AAS, for example, an agent cannot decide to take another route to a certain item. To illustrate the importance of the linkage between the engine and the agents, this part constitutes over 50% of the entire agent code.

On the second and third levels of the architecture, the information from the AAS can be used to check whether the bot's goals are reached or how far off they are. Depending on the character that a bot plays, the fuzzy logic control determines which of the possible paths the bot should start navigating.

Little communication between agents takes place in a normal game of Quake III; it is only used to assign roles in team play situations. Communication is implemented by using the chat system for sending simple text messages. More cooperation between agents would require improved communication facilities. Moreover, currently it is assumed that communication is always successful, which is usually not guaranteed in realistic multiagent scenarios.

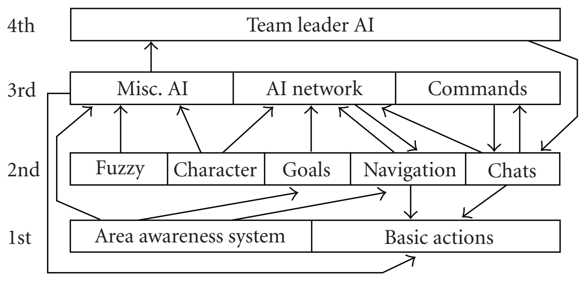


Figure 1: The Quake III bot architecture as described in the developer documentation. This figure shows the close coupling between the various levels of abstraction (Copied from [28]).

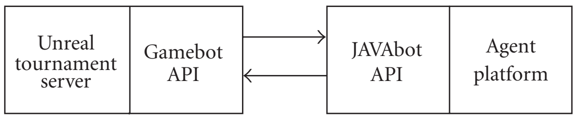


Figure 2: The Gamebot architecture showing that a server-side module is needed to translate the game data to terms the agent can process.

2.2.2. A Client-Side Approach: Gamebots. Gamebots [8] has been created as a research platform for making the connection between agent research and a computer game, namely, the Unreal Tournament environment, and is one of the most used client-side implementations. In client-side approaches, agents are running as completely separate programs from the server and are usually communicating through network sockets. Network communication between agents and other external software programs has been successfully used in other multiagent systems. Gamebots was designed for educational purposes, and therefore, multiple client implementations have been created, for example, one using the scripting language TCL, a SOAR bot, and a JAVA-based implementation.

Figure 2 shows a diagram of the different Gamebot modules in combination with the JAVAbot extension. The Gamebot API forms the extension to Unreal Tournament that is needed to connect a client-side program to the Unreal Tournament. The JAVAbot API is the client side of the coupling. Having a general JAVA API on this side facilitates the connection to most agent platforms because they are usually also JAVA based. Information is sent from the game engine to the agents through the Gamebot and JAVAbot APIs by two types of messages: synchronous and asynchronous messages. The synchronous messages are sent at a configurable interval. They provide information about the perceptions of the bot in the game world and a status report of the bot itself. Asynchronous messages are used for events in the game that occur less often and are directly sent to the agent (but do not interrupt the large synchronous message).

Gamebots is actually not a pure client-side solution because the server is also modified to supply a special world representation to the bot. There are some pure client agent implementations, but they are usually only created for cheating purposes. In this case, the processing of the data is done in the bot itself because it pretends to be a human client game. Doing this filtering on the server is more efficient because only the useful information needs to be communicated. This server modification, the Gamebots network API, performs a similar task and for similar reasons as the area awareness system in Quake III. This clarifies why the Gamebot API is specific for Unreal Tournament; it needs to know the internal representation of the game world in order to make the translation (efficiently).

The Gamebot API does not provide information about the complete environment, but only about objects that are perceivable by the bot. Thus, if a bot wants to gather information about the complete environment, it has to (physically) explore it. To navigate, for example, the agent receives information about predefined navigation nodes in the game map, but only the currently observable nodes are returned. The agent thus does not know what exists around the corner, let alone that it can reason about it. Due to the representation choices made in Gamebots, information about the environment has to be stored at the agent side of the system. This results in large differences between the agent's representation of the environment and the actual environment of the game engine. For complex bots, the information provided by the Gamebot API quickly becomes too limited to make intelligent decisions. For example, the agent cannot know the spawning location of a certain power-up, and therefore, it cannot plan to go there.

There is no facility for communication between agents in the Gamebots API because Gamebots was not designed for adding a multiagent system with interacting agents to the game. It is allowed to add multiple agents to one game, but there are no facilities for direct interaction between these agents. It is possible to create a separate communication system between the agents by bypassing the API and the engine. However, this solution is not only inelegant, but also restrains the game environment to have any influence on the communication. An advantage of separating the game engine and the agents in different processes is that there are no strict time limits on the reasoning process of the agents. A disadvantage of using a fixed API is that the agent receives information it does not need and it cannot access information that it might need.

2.3. Multiagent Interaction. In the previous paragraph, we have seen two examples of ways to connect agents to games. These approaches are limited to a technical way of connecting agents to a game. On the level of game design, few games have tried to leverage these approaches from the start of the game design to add multiple agents and create a more compelling game play. Current games are generally not created with multiagent interaction in mind; interaction is not implemented at all or added as an extra feature in a later phase. For games in which interaction is simple, this is not problematic. For example, Quake III has a game mode in which two teams strive to capture each others flag. The player plays one character in a team, while all the other characters, from his own and the opposing team, are computer-controlled agents. Although the interaction between agents and between agents and the player is limited, the game conveys the feeling of a dynamic interactive world. The same can be said about the communication between characters in F.E.A.R. Although the communication looks quite natural it is actually added to the interaction scene afterward. It thus serves more to enhance reality than that it has a function in the gameplay! See [3] for a description of the problems encountered.

If the interaction in a game becomes more complex and the multiagent interaction is not an intricate part of the design process, some unexpected or unbelievable behavior might occur. For instance, users who were testing the game "Elder Scrolls: Oblivion" by Bethesda games [29] noticed that if they gave one character a rake and the goal "rake leaves" and another a broom and the goal "sweep paths" this worked smoothly. But when they swapped the items, so that the raker was given a broom and the sweeper was given the rake, in the end one of them killed the other so he could get the proper item. If the communication between agents in this game would have been possible, they could have communicated about their goals, and solved their problem. In the academic community, much work has been done on sharing, exchanging, and rejecting goals [30]. So far, this has not been absorbed by the game developer community.

Current games also do not facilitate multiple agents requiring complex decision making. In order to generate agent behavior, complex computation may be required. For instance, in a real-time strategy game, an opponent agent needs to observe the playing field, assess the state of his own units, make an assessment of the strategy of its opponents, generate a strategy, form a plan to execute that strategy, coordinate plans with other agents within the same faction, and in some cases evaluate actions in order to learn from them for future battles. Depending on the algorithms used, this can take considerable processing time. Current games make high demands on computer processors in order to display graphics, simulate physics, create 3D audio, and perform network communication, amongst others. Many games are, therefore, forced to minimize the processing time used for individual agents. If each agent has its own reasoning process running in parallel to generate behavior, this can spiral out of control quickly. This is certainly the case in games with many characters in a scenario. Current games, therefore, often forego the generation of complex behavior and script the behavior of nonplaying characters. For instance, in a first-person shooter game, two computer-controlled players happen to be within equal distance of a power-up. In the current game AI design approaches, such players enter a scripted line of reasoning, resulting in the decision to retrieve the power-up. This will lead them toward the same area in the game and within the shooting range of each other. This behavior is an example of nonrealistic behavior due to oversimplification in a script.

A human player expects the entire game world to persist even when not present in a particular area. Many games have an optimized design that allows a game to be compressed to events, behaviors, and reactions that directly surround the player, and therefore, only the ones visible to the player. So when a nonplaying character falls out of the scope of the player, the game engine no longer simulates the interaction between a nonplaying character and its environment. Thus the game is optimized and the demands on computer hardware are reduced. However, simulating only parts of the game world might result in unrealistic behavior. For instance, in large first-person shooter games, the positions of guards are reset (or their behavior no longer updated) when the player has reached a certain distance. Each time the player returns to the initial area, the guards will be at the same places or even have become alive again while they were killed before. The example shows that simulating an agent depending on the position of the play can lead to discontinuities in the game world.

Conclusion. Most state-of-the-art games use a server-side model with tightly integrated agents. As we have seen, this approach restricts the reasoning time of the agents considerably. An asynchronous solution is more suitable and will be used as a starting point in the next sections. Translating the raw game data to information more suitable for the agents is done in most computer games, but usually in a very restrictive way. In Section 5.2, we propose a more flexible solution. Many of the current games do not use communication at all, and if they do, only for simple tasks and in an ad hoc fashion. Modern games are not created with multiagent interaction in mind. This results in games without or with very simple interaction, or in unexpected behavior in more complicated scenarios. We propose to make the agent interaction an intricate part of the whole development process.

3. Using Intelligent Agents

In this section, we describe some applications of (serious) games that really leverage intelligent agent technology in a way that is currently not practiced. These examples serve to illustrate the usefulness of our approach. The examples in Section 3.1 mainly focus on problems related to information representation. Section 3.2 about multiagent systems stresses the importance of finding solutions for communication issues. The area of synchronization is addressed throughout the whole section.

3.1. Serious Gaming. Besides the purpose of entertainment, games are also used for training and education. These so-called serious games are for example used for the training of pilots, soldiers, and commanders in crisis situations. The training scenarios often involve complex and dynamic situations that require fast decision making. By interacting with these games, the player learns about the consequences of his actions from the reactions of the environment and other (nonplayer) characters to his behavior. Explanations can enhance the player's understanding of a situation [31]. Several approaches of self-explaining agents have been proposed [32–34]. In addition to performing interesting behavior, such agents are able to explain the underlying reasons for it afterward. By understanding the motivations of the other characters in the game, the player learns how his behavior is interpreted by others.

An example of a serious game to which explanation capabilities could be added is virtual training for leading firefighters. In such training, the player (training to become a leading firefighter) has to handle an incident in the game, and is surrounded by virtual characters representing his team members, police, bystanders, or victims. A possible scenario is a fire in a building. During the training session, the player commands his team members to go inside a building and extinguish a fire. The player's team enters the building, but after a while he still does not see the fire shrink from the outside. To better understand the situation, he might ask the virtual characters to explain their behavior. Their possible answer is that they saw a victim inside the building, and decided to save the victim first before extinguishing the fire.

The scenario just given is described on a high level. The virtual characters get commands from the player such as *go to the building, find the fire*, and *extinguish the fire*. When they explain their behavior, they refer to abstract concepts such as priorities between different tasks (saving a victim has priority over extinguishing a fire). However, the abstract decisions that the characters make result into actions that have to be executed and visualized in the virtual environment. Instead of the description *go to the building*, more specific information is required on the implementation level, for example, the coordinates of the agent's starting position, exact path, and final position. So in order to perform actions in the virtual world, the high-level descriptions generated by an agent's reasoning process have to be translated to low-level descriptions required by the game engine.

Besides acting in the environment, agents sense their environment and information goes from the game engine to the agent. The low-level information that is made available by the engine is not immediately useful to the agents. Instead of the exact positions of all the entities and objects in the game at every time step, agents use abstractions, for example, someone is going inside a building, exploring a building takes some time, and the entity in the building is a victim who needs help. The low-level information provided by the game engine needs to be translated to concepts that are useful for the agent. For instance, information about the course of the coordinates of a character could be translated to the more abstract description that the character *enters a building*, and if a state holds for a certain amount of time steps, this could be translated to the high-level concept *for a while*. This concept has to be flexible, as the agent might decide to take an action at time "*t*," but the game engine can only process its action a few steps later. After translating the available low-level information to concepts that agents use, an agent itself can select which of the high-level information will influence its future actions.

For the generation of explanations about agent behavior, a high-level representation of the agent's reasoning process is needed. For instance, agents implemented in a BDI programming language appropriate for the addition of explanation capabilities. Concepts such as goals, beliefs, and plans are explicitly represented in BDI agents and thus available for reasoning and the generation of explanations. Moreover, it has been demonstrated that BDI agents are suitable for developing virtual nonplayer characters for computer games [35]. A nonplaying character however needs to act in and sense its virtual environment, in which other representations of the game world are used. The example illustrates the need of a middle layer in serious gaming, where a translation between the two representation levels takes place.

3.2. Multiagent Systems. Multiple intelligent nonplaying characters bring additional challenges to game design. Currently there are few facilities that allow efficient multiagent behavior. Issues that should be addressed are for example how an agent determines whether there are other agents in the game. If so, how can it communicate with these other agents? How does it know that a message has reached the intended agent? How is information filtered such that it allows an agent to reason about social concepts, for example, about groups, group goals, and roles within a group?

In the firefighting scenario sketched in the previous subsection, the team members of the leading firefighter (player) are intelligent agents (nonplayers). Although they have to execute the commands of the player, they still need intelligence of their own. In the first place because they might take initiatives by themselves; in the scenario the nonplaying characters decided to save the victim first instead of extinguishing the fire as the commander had told them. Second, because they act in a team and have to coordinate their actions with each other. For instance, if the group has to decide whether to go left or right, they have to communicate to each other in order to make a common decision. Or, only one of the characters needs to carry an axe for opening doors, but the others have to know that one of the team members is responsible for this task.

Suppose that a team of firefighters goes into a building with the goal to extinguish a fire. One of the members is responsible for opening locked doors and another has to extinguish the fire. If the first carries an axe and the second an extinguisher, this will work smoothly. However, the situation in which the door opener carries an extinguisher and the fire extinguisher and axe is more complex and requires communication. The door opener has to be aware of the other character, come up with the idea to communicate with it, send the right message, wait—long enough—for the result, and finally connect the right action to it. The next action of the door opener depends on the information it receives from the fire extinguisher.

We believe that the communication between different agents in a game should go through the game engine instead of taking place on the agent platform because the effect of communication has influence on the game world itself and not only on the agents. For instance, if the two agents in the scenario successfully communicated and decided to exchange their tools, this needs to happen physically in the virtual environment as well. If communication would not go through the game engine, there is a danger that processes in the game world and between the agents are no longer synchronized. For example, if the agents agree to swap items, they would both send a message to the game engine and believe that the items will be successfully exchanged in the game world. This however is not obvious. The actual swap in the virtual world is managed by the game engine, for example, one agent puts down its tool, has its hands free to receive the other tool, and the other agent picks up the tool from the ground. For such a process, it is crucial that the game engine receives the messages from both agents at the same time, or at least connects them to each other. This can be better realized if the game engine is included into the communication loop.

In turn, physical changes in the world have effect on communication as well. For example, if the door opening agent asks a team member to take over, it expects this member to come and take his axe. By perception, the agent derives whether its colleague perceived the message and decided to assist, or if it should communicate more. The colleague might have a good reason to refuse, for example, it has to assist a third agent already. It could communicate this to the requesting agent. The timing of this communication and the action to help the third agent should be synchronized; otherwise the requesting agent might for example unjustly belief that it is being ignored. Such timing is facilitated by including communication into the game loop.

Further issues concerning careful time management include a translation of time for the game engine to time for the agents. For instance, if the door opening agent sends the message *what tool are you carrying?* To the other agent, it expects a reaction. It is not realistic to expect a response directly in the next time step, the game engine could give priority to other processes first and the other agent might need some time to reason about the question. However, the agent also should not wait indefinitely because it could be that the message never arrived, or that the other agent misunderstood the content, and so forth. So after a certain amount of time, the agent has to react, for example, by sending the same message again, or by sending a message *did you understand my previous message?* In the middle layer, a translation of time for the game engine (a number of time steps) to time for the agents (time in which a reaction could be expected) has to be made.

The examples in this subsection aim to make clear that communication is more than just an exchange of information. After sending a question or a command, the sender expects an answer or action. If it does not see an effect of its communication action for whatever reason, the sender will react on that. Decisions of agents depend on the information they receive by communication *and* perception, and their communication actions have effect on the game world *and* the behavior of other agents. Therefore, the communication processes and the actions in the game world have to be well synchronized.

4. Connecting Games and Agents, Our Vision

In Section 2, we have shown current approaches to integrate agents in game engines. It is clear that those solutions are pragmatic but do not really give room to fully use all aspects of agent technology in the game environment. In Section 3, we have illustrated how agent technology can contribute to the use of game for serious purposes and a more compelling interaction between NPC characters. To overcome issues with synchronization, information representation, and communication, we analyze the connection between game and agent technology from three different perspectives, that is, the infrastructural, conceptual, and design perspectives.

For our solution, we look at the connection between the agents and the game engine starting from infrastructural point of view. The main requirement is that on the one hand the game engine should have some control over the actions of the agents in order to control the overall game play and preserve physical realism. For instance, if an agent wants to move in a straight line to a position in the game world, but there is a wall in between the agent and that point, then the game engine will prevent the agent from moving to the point it wants to get to, that is, the agent cannot just move through walls. On the other hand, the agents should be autonomous to a certain level. For instance, if an agent is walking to a way point, but is reconsidering his decision and wants to turn back, it should not first have to walk to the way point and only there be able to turn back. Also, we want the agents to be able to keep reasoning full time and not being restricted to specific time slots allocated by the game engine.

An important consideration in the connection between the agents and the game engine is which information is available to the agent and when and how does it get that information. Moreover, we have to consider when agents can perform actions in the game and which actions are available to the agent. With respect to the latter, one should think more in terms of abstractions than in terms of forbidden actions. For example, can an agent open a door or should it manipulate a door object position to another position? Often the translation between these types of actions is provided for the avatars steered by the user. However, it is not clear that the same set of translations applies for the nonplaying characters in the game. For example, current animation engines are capable of performing rudimentary path planning. This means that actions become available to characters to move through a room without bumping into any object with one command. These commands might not be available for the human players, but are very efficient for the nonplaying characters.

The above considerations all relate to the connection of a single agent to the game engine. In general, one would like to connect a complete multiagent system to the game in which the agents also can communicate and coordinate their actions. In order to fully profit from agent technology, one would want especially to have the agents using their own high-level communication protocols that facilitate coordination. These communication facilities are standard provided by the agent platforms on which the agents reside normally. As we have seen, the facilities for communication within the game engines are rather primitive and/or ad hoc. So they are not very suitable for this type of communication, unless we extend them considerably.

The next question thus becomes how to connect the agent platforms to the game engine. Several solutions are possible. First, one can integrate the functionality of these platforms in the game engine. In this case, the agents can be built as if they are running on an agent platform. Second, one can distribute the functionality over the game engine and the agents. This means that some rudimentary functionality is incorporated in the game engine, but the agents have to get some more elaborate communication functionalities to compensate for the loss of some features. For example, they might have to keep track of the other agents they can communicate with (storing agent names and addresses). A last option is to let the agents run on their own platform and connect the platform to the game engine. One problem with this option is that the platform runs in parallel to the game engine and all types of interactions between the agents are not available to the game engine. This might potentially lead to a loss of control and inconsistencies between the agents and the game engine.

We will opt for a position in the middle. We will transfer some of the communication functionalities to the game engine to preserve consistency and control. However, we also will keep the agents running within their own platform. This is mainly done for some other facilities provided by the platforms, such as efficient sharing of reasoning engines by the agents and monitoring and debugging interfaces for the agents. The last parts are important for designing and implementation, but can be decoupled in the runtime version of the game. In order to address all issues, we divide the connection into three stances: an infrastructural, a conceptual, and a design stance.

As indicated above, the infrastructural connection requires adjustments on both the agent as well as on the game engine side. Therefore, although the connection principles might be platform independent, the actual implementation will not be completely platform independent. The standard way to ameliorate this point is to create a middleware API. Basically, connecting agent (platforms) to game engines is not different from connecting any other software together. So, in the end, we also will make use of the means available to connect independent threads of software. However, what is different is the perspective. In most applications that connect software, there will be a single thread of control that is well defined. In our case, we want a kind of shared control that is different from traditional software solutions. It means that our infrastructural solutions should take this perspective of shared control already in mind and be as flexible as possible in order to define the way control is shared on higher levels. So, in our middleware, one can define the standard constructions that we assume to exist on both sides, but the way they work together is kept as flexible as possible. The exact sharing of control is defined in the infrastructural stance. We describe the infrastructural stance in more detail in Section 5.1.

The translations between information representations that are needed to connect the agents to the game are described using a conceptual stance. Most important will be the translation of actions of the agent into actions within the game engine and translations of changes in the world into percepts that can be handled by the agent. We aim to use the high-level architecture (HLA) standard for this purpose. This stance is described in Section 5.2.

Finally, it is important to incorporate the agents explicitly in the design method of the games. The type of data that has to be generated or kept depends crucially on the ways that the agents need to use them. Therefore, if the world is first created and the agents are only added in the end, they might not have enough information available to act intelligently. For example, if an agent has to take cover it should know the distinction between an iron bar fence and stone wall of the same dimensions. If the only data available is that there is an obstacle of certain dimensions, this information can hardly be deduced. Designing the environment with the possible actions and perceptions of the agents in mind will drastically change the way the world is created. In Section 5.3, we will show that the agent-oriented OperA framework is a good starting point for such a design methodology.

In Table 1, we summarize how the different issues that we focused on are dealt with within the different stances. In this table, we denote the technique that is used in a particular stance to deal with an issue. Please note that the issues are not all of the same type. Synchronization, for example, is a technical issue that is, therefore, not really discussed in the design stance. In contrast, communication is such a general issue that it has elements that are dealt with in all the different stances.

5. Three Stances to View the Connection

In this section, we will discuss the three stances (infrastructural, conceptual, and design stance) in our approach more extensively. For each of them, we will indicate their contribution to gaming scenarios as described in the previous section. As argued before, the topics of synchronization, information filtering, and communication play a fundamental role in coupling games and agents. So they all will be covered in this section as well. Synchronization is mainly addressed in the subsection about the infrastructural stance. Information filtering receives most attention in the subsection about the conceptual stance. Communication involves several aspects; it is, therefore, discussed in all of the three subsections.

5.1. Infrastructural Stance. In our approach, we view the game engine and agents as asynchronous processes because, as discussed in Section 2, agents that are part of the game loop are restricted in their reasoning by time. Therefore, we believe that a synchronous approach is not suitable for intelligent agents with complex reasoning processes. Although we are investigating a coupling between two specific types of asynchronous processes, infrastructurally our case is similar to other asynchronous couplings.

There are four basic tasks that need to be performed by the infrastructure. First, information about the game environment needs to be provided to the agents to allow them to reason about the game. Second, the actions that the agents have selected to perform in the game need to be transferred to the game engine to allow them to be executed. Third, the communication between agents can be effected by the game environment and thus needs to flow from the agents, through the game engine, back to the agents. Last, the infrastructure needs to provide a central time. The latter is relatively simple and done by sending timed events to both game engine and agents.

When an agent requires information from the game engine, a distinction is made between information about static and dynamic game entities. Static entities have properties that are fixed for the duration of the game, for example, buildings, mountains, and roads. Dynamic entities contain properties that change continuously. For example, victims have changing health, firefighters change position, and fire spreads through a building. For static entities, the engine can inform that the entity is static and include the requested properties. After such a message, the agent normally does not need to update this information anymore. This thus provides for some efficiency in the information flow.

For dynamic entities, the game engine sends a message when entities become (un)perceivable for the agent. The conditions for perceivability are defined conceptually. In the filtering layer is decided which events are relevant for that specific agent. This in contrast to fixed APIs used in current work where all agents receive the same event types. After being subscribed to a dynamic entity, the game engine will keep sending updates about these properties. This mechanism prevents the agent from being flooded by information about all possible entities and their properties, while not limiting it to predefined aspects of the game world. One could see the decision-making process that selects which events are selected as part of the agent but this is not a necessity.

Table 1: Contribution of each stance to the three challenges of connecting agents to games.

	Infrastructure	Conceptual	Design
Synchronization	Event queues	Timestamps	—
Filtering	—	HLA	Ontology
Communication	Communication queue	ACM	Interaction patterns

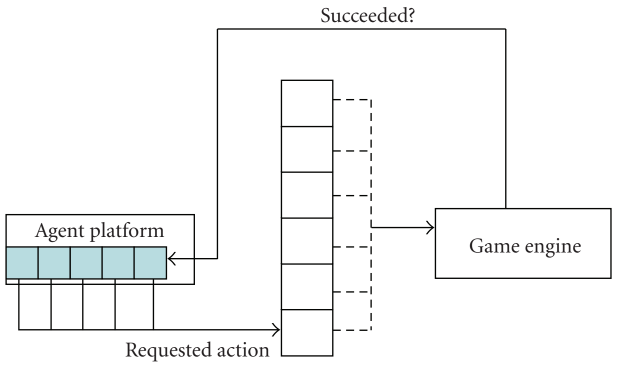


Figure 3: Event queues in the infrastructure allow both the agent platform and the game engine the flexibility to select events based on their own criteria.

In order to execute actions in the game world, an agent sends a request to the game engine. However, the agent's actions might be of a different type than the game engine's actions, for instance, open a door versus move object x to position y, z. Moreover, the timing of the request might not correspond to the game loop, so directly executing these actions in the game engine is in general not possible. Therefore, we propose to implement the actions of the agents in the game world by a queue structure which contains a description of the action plus possible timing constraints. Figure 3 shows a diagram of the information flow of the action requests. A requested action is inserted at the end of the queue to keep an ordering of the requested actions. At the beginning of each new time step of the game loop, the engine selects actions to perform. One possible approach would be to always select the actions at the top of the queue. The game engine however is able to select actions based on other criteria. For example, certain actions might be preferred by the game engine or a higher priority might be given to actions by a certain agent.

Normally agents expect an external action to behave like a method call and the agent waits for the result. But because actions in the game world are not always executed right away, do not always succeed, and sometimes have unexpected results, we separate the request of performing the actions from the result. The result of the execution of the actions is sent back from the game engine to the agent in a separate message. Agents do not have to stop their reasoning process to wait for this message. When the message arrives, the information can be used for further reasoning. This is significantly different from normal multiagent programing. There are different ways to cope with this delayed feedback. There is no guarantee that the engine responds within a certain time limit. An agent could be programed in such a way that it assumes that the action failed if no response is received within a fixed amount of time. Or an agent can assume that all actions succeed and if a negative response is received from the engine, this information is corrected. More elaborate reasoning about this information is also possible.

When comparing this approach to the Gamebots model, it is obvious that there are some similarities. Gamebots also uses separate asynchronous processes and some adjustments can be made on the timing of information passing. However, there are some advantages to using the approach suggested in this section. The main difference is that there is a lot more flexibility on the kind of information that is passed from the engine to the agent because of the usage of a subscription model instead of the fixed API. For example, in the subscription model, the agent could subscribe to a very specific property such as the health information of another character. With the fixed information passing used in Gamebots, all predefined information is continuously sent, and therefore, such specific properties are omitted. The timing is also more flexible on both sides of the system. When using Gamebots, information is either sent to the agent at a fixed time interval or directly for synchronous or asynchronous, respectively. It is not possible to change the timing or the amount of information that is sent, although the interval between messages is configurable. In the subscription model, the agent can request information whenever it is convenient. On the game side, the engine selects the requested actions from the input queue at its own time and with its own selection criteria. In the Gamebots approach, the actions requests all have to be executed immediately in the next time step.

Communication between agents is organized in a similar way to action requests and information passing. Sending a message to another agent is treated as a type of action request, where the action consists of delivering information to another agent. Communication plays an important role in multiagent systems, which is why we prefer using a separate queue for communication requests. Again the game engine can have its own preferences about the selection of messages from this queue. The game engine determines how a communication request is handled. For example, if an agent shouts, the engine determines which agents receive this message. Similar to action requests, the agents also receive feedback about the result of sending the message. Because the game environment can influence the success and effect of the communication, it is clear that it should pass through the game engine and cannot be organized through the multiagent system platform (as is normally done).

5.2. Conceptual Stance. The second stance in our framework connects the character's mental capabilities (implemented in the software agent) to their physical counterpart (implemented in the game engine), in a similar fashion as the pineal gland was supposed to connect the mind and the body in Rene Descartes' dualist worldview [36]. In this section, we will discuss the mapping of agent reasoning symbols to game engine data.

5.2.1. Conceptual Agreement. The most important aspect of this stance consists of a translation between concepts in the game engine and the agent. For example, when an agent wants to execute the action *go to the building*, this should be translated in the game engine to *find object called building*, *check object can be reached*, *plan path to object*, *follow path to object*, and vice versa. In order to create this mapping, we need to define a consistent common representation. This representation functions as an agreement or contract between the game engine and the agent. While each has a different internal representation of the concept, both have to respect the meaning of the concept as defined in the agreement.

The agreement will cover the way the world can be perceived by the agent (game engine to agent mapping) and the way the world can be acted upon by the agent (agent to game engine mapping). These agreements are called the object perception model (OPM) and the object interaction model (OIM). They are inspired by HLA. HLA is a simulation interoperability standard [37]. HLA was designed to allow different simulations to connect and participate in a shared scenario. However, it was not designed to connect agents to simulations or games. One aspect that is required for the case of connecting an agent is filtering of data. An agent should only receive data that is relevant for the agent. For example, if an agent is fighting a fire in a building, it is of little use to receive a message that there is a player on the other side of the game world that lost his helmet. In HLA, there is only control over data distribution among participants by the use of a publish-subscribe approach. However, in the case of agents, the need for information is highly dynamic and based on the situation at hand and the line of reasoning by the agent. Therefore, the condition under which subscriptions should change needs to be represented. In the case of agents, we will extend the HLA approach with more control over data distribution. This extended control will create a more dynamic publish-subscribe approach in which a party is only subscribed to certain information in relevant situations.

First, we will describe the object perception model. The OPM represents both the entities that can be perceived (ontological representation) and the condition in which they can be perceived (qualification representation). In HLA, the common ontological representation is defined in the federate object model (FOM) which is an instantiation of the object model template (OMT). In the case of agents, we will not need many of the data types defined in the OMT and we can use a general syntax such as XML. For example, a firefighter in our scenario can observe other characters. The following XML description indicates which features of the characters it can perceive:

```
<class name="Character">
  <property>
    <name>ID</name>
    <type>number</type>
  </property>
  <property>
    <name>Distance</name>
    <type>meters</type>
  </property>
  <property>
    <name>Direction</name>
    <type>Orientation</type>
  </property>
  <property>
    <name>Tool</name>
    <type>Tool</type>
  </property>
</class>
```

Stating that one can perceive the character's distance and the tool it carries. An agent can for instance subscribe to perception messages about other characters. Only when it is relevant should the agent actually receive these messages. This is accomplished with the Poss() operator. It means that only messages will be sent when the situation satisfies some constraints. For instance, the conditions in which the characters can be perceived can be described as follows:

$$\begin{aligned}&\mathrm{Poss}(\mathrm{Perceive}(\mathrm{Character}, \mathrm{ID})) \Longleftrightarrow\\&\ (\mathrm{Dist}(\mathrm{Character}, \mathrm{ID}) < 150 \wedge \mathrm{LineofSight}\\&(\mathrm{Character}, \mathrm{ID}) \wedge \mathrm{Direction}(\mathrm{Character}, \mathrm{ID}, \mathrm{towards})\end{aligned}$$

So, a character can only be perceived if it is closer than 150 meters and one looks in the right direction. Both the agent and the game engine need to interpret the OPM based on this common representation.

In the case of the game engine, the part of the game loop that sends world data to the agents contains a list of agents that have the capability "perception." This capability is described in the OIM (which we will introduce hereafter). It also contains a list of objects of the type "Character." It compares the x, y, z positions and checks if the distance is smaller than 150 meters, checks if there is a line of sight between each agent and each character and checks the relative orientation of agent and character. If these actions satisfy the perception rule in the OPM, the game engine sends an asynchronous message to the queue of the perceiving agent. The message contains the "Character" object with the properties as defined in the OPM.

The mapping of concepts between game engine and agent can be facilitated by software tools that automate some of such mappings. For example, Kynapse from Kynogon [38] is able to analyze geometric data and extract path planning information from this data. This in fact is an automated step to translate game engine information to concepts with which an agent can reason. For information other than path planning, additional tools could be developed.

In the case of an agent created in an agent language such as 2APL, it will interpret the OPM straightforward as an incoming event of the type perceive with a number of parameters.

Event (Perceive (Character, ID, Distance, Direction, Tool), TIMESTAMP). The timestamp indicates the time the event was received. These types of events are stored in the so-called event base of the 2APL agent. It can use reasoning rules to decide what to do with these perceptions. For example, it can update its belief base each time such an event is received, but it can also restrict updates to characters that are closer than 50 meters or of which the distance changed more than 100 meters.

Second, we will describe the object interaction model. The OIM represents the capabilities of the agent to interact with the world. It denotes the possible interactions, the conditions under which they are possible, and the effects of an action. In HLA, interactivity between simulations is achieved through sending specific interaction events. These interactions are messages specifying events that happen in a simulation. Based on the subscriptions of a simulation, it will receive all corresponding events. In the case of agents and games, we again need more precise control over which interactions are relevant for an agent. This helps reduce processing load on the agent side and optimize the game on the game engine side. We again take inspiration from HLA and define the interaction relevant properties of objects using XML. We then extend this with rules specifying constraints and consequences concerning the actions. We continue the firefighter example and describe an agent (which can be viewed as an object that can interact) that can open doors:

```
<Agent name="Door-opener">
  <general>
    <property>
      <name>HoldsOpeningTool</name>
      <type>Tools</type>
    </property>
  </ general>
  <physical>
    <property>
      <name>height</name>
      <type>meters</type>
    </property>
  </physical>
  <sensor name="eyes">
    <property>
      <name>Range</name>
      <type>meters</type>
    </property>
  </sensor>
  <capability name="Open door">
    <property>
      <name>target</name>
      <type>Door</type>
    </property>
  </capability >
</Agent>
```

We force agreement on the circumstances before and after the action in a similar fashion as done in [39] by specifying pre- and postconditions of the action:

$$\begin{aligned}&\text{PRE: Poss(OpenDoor(Agent, Door))} \iff\\ &\text{Closed(Door)} \wedge \text{Distance(Agent, Door)} < 1\\ &\wedge \text{Holds(Agent, Axe)}\\ &\text{POST: Done(OpenDoor(Agent, Door))}\\ &\Longrightarrow \text{Open(Door)} \wedge \text{Poss(Backdraft(Door))}.\end{aligned}$$

So the agent can open a door if the door is closed and it stands near to the door and is holding an axe. If a door is opened, its state is changed to open and the agent is automatically subscribed to messages that indicate a back draft explosion occurred.

Similar to perception, both the agent and the game engine will need to interpret the OIM. For the game engine, this means that the "Door-opener" agent will be subscribed to asynchronous messages (as described in the previous section) about "Door" objects in its vicinity. This is interpreted from the appearance of door objects both in the agent properties and in the interaction rules. Additionally, the game engine processes code to execute "OpenDoor" actions sent by the "Door-opener" agent (i.e., changing the status of the door to open) while it ignores such actions from other agents. It changes the physical representation of the door by turning it ninety degrees. Following this, the game engine recalculates fire and heat intensity and the oxygen level in the room behind the door. If a door is opened in a room that is very hot but contains little oxygen, the game engine will produce a message indicating that a back draft explosion occurred.

The link to the agent side has to be made through the capabilities of the agent. In 2APL, this is an easy process because the capabilities of an agent are given explicitly in the agent program with their pre- and postconditions. For example,

```
{Closed(Door), Distance(Agent,Door) < 1,
 Holds(Agent,Axe)} OpenDoor
{Open(Door)}
```

By forcing agreement on the concepts used between agents and the game engine, each can have their own internal representation while there is an agreement on what can be communicated and on what level of abstraction.

5.2.2. Communication. In a multiagent setting where agents need to coordinate their actions, they must communicate. Communication between agents can be achieved in similar fashion as actions and perception of the agent. The action of an agent now is the sending of a message, while the perception consists of the reception of a message. Sending a message consequently requires describing the pre- and postconditions. Receiving a message is controlled by an agent subscribing on messages and by the game engine when it determines that an agent can sense an action. In this case, we do not only specify the agreement between agent and game engine, but also between agents. So there are three or more parties that need to conform to the agreement instead of two in the previous case. We will call this agreement the agent communication model (ACM).

The definition of the ACM will contain the type of things that can be communicated (communication content representation) between agents. This representation is only relevant to the agents in the game. The ACM also specifies when communication can take place (qualification representation). This specifies the impact of the environment upon communication. For instance, if an agent is far away, it may not be able to communicate. These factors are relevant for the game engine that is responsible for simulating the environment.

There already exists a formalism that provides a communication content representation. It provides a way to communicate such things as beliefs among agents or propose an action or communicate with multiple agents. Within the FIPA standard [17], these communicative acts are already defined. We propose to use the FIPA standard to establish a game-specific message structure. For example, in our firefighting game, agent A may propose to agent B that A opens the door to the building:

```
(propose
        :sender (agent-identifier:name A)
        :receiver (set (agent-identifier:name B))
        :content
                "((action A (open door))"
        :ontology Fire-fighting
        :in-reply-to proposal2
        :language fipa-sl)
```

The message is translated to the concepts internal to both the agent and the game engine. The game engine will, upon reception of the above message, send this message to agent B and automatically subscribe agent A to the communication messages of agent B. This is because agent A and B can now be said to be in a dialogue and it is likely that agent A would like to receive an answer. The game could progress such that agent B replies affirmatively and the game engine receives an action from agent A to open the door and an action from agent B to go through the door. The game engine will now have enough information to know that this is a coordinated action and that the order of actions (as implied by the dialogue) is to process the door opening action first and the movement action second. The game engine therefore takes these messages from the incoming actions queue and processes these together (coordinated) and in the right order.

To describe the impact of the environment on communication, we have to augment the linguistic representation with information about the environment. Since communication is a form of action, the same qualification representation needs to be made explicit. These qualification rules will also need to specify the ramifications of communication. This allows us on the one hand to specify what is needed when agents want to communicate (e.g., that they are close together) and on the other hand the (side) effects of communication (e.g., if other agents than the message recipient are nearby they too may receive the message):

$$\begin{aligned}&\text{PRE: Poss(Send(Propose(Action,Agent)))}\\&\quad\Longleftrightarrow \text{Dist(Agent)}<5\\&\text{POST: Done(Send(Propose(Action,Agent)))}\\&\quad\wedge \text{Dist(Agent')}<5 \Longrightarrow\\&\quad\text{Poss(Receive(Propose(Action,Agent)))}\end{aligned}$$

5.2.3. Time. Time is an important aspect in the connection between game engine and agent. Both need to agree on a reference of time. In our approach, the game engine provides the time by sending periodic time messages to all agents. The meaning of these time messages is defined in the OPM:

```
<class name="time">
  <property>
    <name>value</name>
    <Type>Seconds</type>
  </property>
</class>
```

The game engine will translate its own data in milliseconds to seconds and send the messages. The agent will translate these time messages into meaningful symbols that are relevant to the agents updating a belief it formed an hour ago to an "old belief" or "stale belief." Additional to these time messages a game designer is free to add additional facilities, such as allowing agents to query how much time passed between two events. Such a service could provide the translation from milliseconds in the game engine to concepts such as "just now," "a while," and "long ago."

The above contracts (i.e., OPM, OIM, and ACM) will be derived from the game design process in the design stance. For instance, it is established that a game interaction takes place in a scene called "building." The game designer can then start to construct the contracts that describe the concepts of that building that are relevant to both agents and the game engine.

5.3. Design Stance. In the previous sections, we discussed how agents could be connected to game engines infrastructurally and conceptually. However, creating these connections does not automatically mean that they are used in a proper way. Game design uses several methodologies [40], but all consider aspects such as rules, play, and culture. We will follow [41] and distinguish the following channels.

(i) *The abstract rules governing the game play.* For example, this determines the strength of weapons or what is needed to open doors, and so forth.

(ii) *The storyline.* This determines the overall narrative. For example, in Quake, the story is about capturing a flag.

(iii) *The user interface.* How is the game environment represented and how does the user interact with it.

(iv) *Look and feel.* What emotions are generated by the game, what kind of feeling it gives. For example, are enemies extraterrestrial beings or soldiers?

Current practice in game design assumes that the human players are intelligent. The game rules are meant to regulate how the users can interact with the game and ensure that the storyline is kept. At this moment, the only place where AI plays a significant role is on intelligent path planning. All characters have to do some form of path planning to get around in the world and this is a nice modular task that can be enhanced by some more realistic or intelligent path planning. Looking at the different channels, we see that it mainly influences the look and feel channel as it makes the characters move more natural. It thus has no fundamental influence on the game play.

This will be quite different when the characters are played by software agents that can be autonomous, adaptive, and intelligent and moreover can communicate with each other. Once these features are added, it is unclear whether the same game rules still ensure the same game play. Once characters can reason about the world, start cooperating and adapting to the players, the game might fundamentally change of character, and it is not directly clear if it will change for the better!

If we want to add software agents that can behave more intelligent and adaptive, we should also design the game rules such that the game profits from this behavior. Thus, we should take the capabilities of the characters already into account when designing the game rules! For example, a game rule that determines that in a firefighter training, at least one of three doors is locked to make the firefighting more difficult becomes useless if the characters learn how to open a locked door as quick as a nonlocked door. This becomes even more apparent if we consider that agents might also communicate (in a more or less unrestricted way). Adding communication capabilities to agents means that they can start to cooperate and thus circumvent some of the rules in the game. For example, one character can start extinguishing the fire while the other saves a victim. The one that goes inside the building to save the victim can be determined by which of the two knows the building better. This can be easily determined through communication, but is hard to preprogram. It does mean that the agents will be able to achieve more than when used independently. This kind of elements should thus also be modeled in the game rules channel. In general, one would

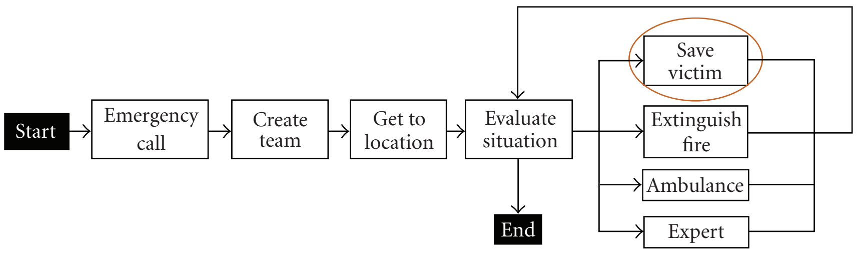


FIGURE 4: Interaction structure in OperA.

have to take into account who can communicate with whom and whether communication always succeeds. In a realistic environment, a character might only be able to communicate by "talking" to characters in the same physical space while other characters are overhearing the conversation.

In multiagent systems, communication mechanisms provide standard ways of dealing with these issues. However, they do not assume the agents operate in a game environment. Thus the mechanisms would have to be adjusted to the game engine.

The above points illustrate that, if we assume that characters are played by intelligent software agents that can communicate, the game rules should be designed in such a way that the storyline will still be guaranteed. Moreover, if we assume the characters to act intelligently, they should also have the means to do so, that is, they should have the right information available at the right time. For example, if a character has to avoid being seen, it does not make sense to duck behind an obstacle which happens to be an iron fence. However, if geometric features of the obstacles are the only available information, it will be hard to create intelligent behavior based on them.

This pleads for the fact that we should take possible intelligent behavior and the requirements for this behavior on the world already into account in an early design stage. One could also argue to start modeling the agents using an agent-oriented software methodology. This at least ensures a proper modeling of the agents and their interactions in the game. However, agent-oriented methodologies hardly take the environment in which the agents operate into account. Therefore, the modeling of the actual world and the intricate interactions that are needed in the game environment are not supported sufficiently.

This leaves only one way open, which is a new design methodology that allows designing the game environment and the agents concurrently. We believe that there are good starting points for creating such a methodology if we use an agent-oriented methodology that also takes the agent's organization into account.

Roughly the methodology should start with designing the game rules and storyline at a high level. At this level, the specific actions that take place are not fixed yet, but only the required landmarks that the game should pass through. In the next stage, the designer should determine which agents would possibly play a role in the different scenes that lead to these landmarks. Note that the order of scenes might still vary. He can then decide which requirements have to be fulfilled by the agents to perform their actions in the different scenes and what kind of information should they have available if they want to exhibit some intelligent behavior. Besides these requirements, he also has to give the constraints on the actions, for example, opening a door requires an axe, to ensure that "intelligent" behavior does not lead to completely unexpected and unwanted behavior.

The requirements on the availability of information lead to requirements on the conceptual contracts. The boundaries on the actions lead to requirements on the capabilities of the agents, for example, the precondition of opening a door is to carry an axe.

5.3.1. A Design Methodology: OperA. We propose to use OperA [42] as a starting point of a framework to model games incorporating agents. OperA provides a model for agent organizations that enables the specification of organizational requirements and objectives, and at the same time allows participants to act according to their own capabilities and demands. It still needs to be extended with a more elaborate environment model to capture the game world aspects. In this paper, we will focus on the specification part of the agents. Role descriptions in OperA define the activities and services that have to be performed to achieve the game objectives. These objectives are distributed over the objectives of the agents. Role descriptions also define the rights and capabilities of the agents.

By clearly defining these capabilities in an early design stage, we can guarantee that they are implemented in the game world (through a conceptual translation). Table 2 shows an example of a role description for an agent of the type "leading firefighter." From this description, the objectives of this type of agent become clear and it already gives some idea about the information needed by the agent to realize these objectives. A part of the game rules is specified by the norms. The rights of the agents also define a part of the game rules and need to be translated to actual capabilities in the game engine.

In OperA, the overall storyline is specified by the interaction structure. The main purpose of this structure is to specify an ordering between separate scenes in the game and to make sure that required states are always reached. The ordering is not always linear; scenes can be executed multiple times. The actors that participate in the scenes of

Table 2: Role definition in OperA.

Role: leading firefighter	
Objectives	Fire_under_control, victims_save
Subobjectives	{get_to_disaster_location, situation_assessment, plan_of_attack, extinguish_fire, rescue_victims}
Rights	Command_team_members, order_ambulance, get_experts
Norms	OBLIGED inform(headquarters, plan_of_attack) BEFORE NOW+10 IF DO safe(victim) or DO extinguish(fire) THEN PERMITTED damage(building) OBLIGED ensure_safety(team) OBLIGED safe(victims) BEFORE extinguish(fire)

Table 3: Interaction scene in OperA.

Interaction scene: save victim	
Roles	Leading_firefighter(1), door_opener(1), fire_extinguisher(1), ambulance(2), victim(3)
Results	r1 = ∀ T ∈ victim, safe(T)

Interaction patterns	PATTERN(r1) =
	{DONE(T, at(H,T)) BEFORE DONE(B, secure_area),
	DONE(B, secure_area) BEFORE DeadlineH,
	DONE(M, stabilise(H) BEFORE Dead(H))
	DONE(T, transport_to_ambulance(H))
	}

Norms	PERMITTED (E, blow_obstacles) OBLIGED (M,stabilise(T) BEFORE Dead(T)) OBLIGED (B, extinguish_fire BEFORE transport(H))

the game and the way they interact are defined in the scene level description. Figure 4 shows a graphical representation of a possible interaction structure. The transitions between the different scenes are specified in the interaction structure to make sure that a scene is entered and terminated in such a way that the storyline is guaranteed.

A scene is a formal description of the interaction space between different agents for a specific part of the game. In these scenes, the types and number of participating agents are defined, and the interaction between the agents themselves and the environment. The result of a scene is specified and optionally norms can be added.

Table 3 shows a possible description of the "save victim" scene from the interaction structure above. For each role that is possibly active in this scene, we specify the number of agents that fulfill that role. For example, there is one leadingfirefighter and there are three victims. Most importantly, the desired results of the scene and the interaction patterns between the different roles are specified. In an interaction scene, separate norms and permissions can be specified that determine specific game rules for the interaction.

Starting from the interaction patterns of the scenes, different messages and other forms of communication can be specified, and a platform-independent design can be created. The ordering of these communicative actions can be strictly defined by a protocol or, more flexible, an interaction diagram. From this platform-independent model, we can move on to the platform-specific phase, in which the agent interface and the interaction specifications are implemented. If certain inherent limitations on the communication are known in advance, these limitations should already be taken into account during the platform-independent design phase. If they surface during the implementation phase, it is usually better to go back and adjust the platform-independent design. In the design phase, the agents' knowledge about the environment and themselves should be modeled. This information can later be used in the platform-specific design to create the data models.

After we specified the agent roles and interactions, decisions have to be made about the agent implementation. The requirements that the agents need to fulfill have to be taken into account in this step. For example, if the agents have to be able to explain themselves [43], it is necessary that they use high-level concepts in their reasoning, such as beliefs, intensions, and goals. A logical decision in case of this requirement would be to implement the agents in a BDI programing language. Another example could be learning or adapting agents; the appropriate agent type needs to be selected to allow for the expected adaptability. Also the learning algorithm itself, the elements that are adapted and the feedback type have to be chosen.

The multiagent interaction also has to be specified more precisely. In the interaction scenes, we already define a high-level definition of the interaction. As we have seen in Section 3.2 (multiagents systems), certain tradeoffs have to be made on the amount of communication. In the design phase, a clear definition has to be made of what information is passed on and when. Designing decisions also have to be made about the activity of characters that are not playing an active role in the current scene.

Besides specifying the technical requirements, some quality requirements have to be kept into consideration as well. An important quality requirement for computers is that the agents and other parts of game respond fast enough. The behavior of agents should be believable. Games should be esthetically pleasing and a certain atmosphere in the design is desired. These quality requirements are mostly related to the look and feel channel and the user interface channel. Sometimes they can be translated to a technical specification, but most of the time they have to be considered during the whole development process without being captured by a precise technical requirement.

6. Conclusion

There is consensus among game developers that intelligent characters for games can make games better. However, there is a difference in the approach to bring intelligence about between the game developers and the artificial intelligences researchers. Consequently, using agent technology in combination with game technology is not trivial. Because agents are more or less autonomous they should run in their own thread and can only be loosely coupled to the game engine. Synchronizing the agents with the game thus becomes an important point. Once the agents are synchronized not all problems are solved. Because agents usually function on a more abstract level than the game world representation allows. A translation is needed between the game world information and processes to the beliefs and actions of the agents. Finally, agents should be able to communicate not only with the game world but also with each other. Thus there is a need for communication mechanisms that connect both the agents and the game world. We have seen that current combinations of games and agents only deliver limited or ad hoc solutions for all these issues.

In this paper, we argue that improving the AI in games by using agent technology to its full extent involves solving the issues above. Furthermore, solving the synchronization, information representation, and communication issues requires more than constructing a technical solution for the loosely coupling of some asynchronous processes. Although this aspect is a fundamental part of the coupling, we also need to provide support on a conceptual and design level. Using a conceptual stance allows for connecting the agent concepts to the game concepts such that agent actions can be connected to actions that can be executed through the game engine and that agents can reason intelligently on the information available from the game engine.

We also argue that coupling agents to games requires a design methodology including agent notions from an early stage in the design process in order to allow a full integration of agent characteristics in the game and to profit from specific agent characteristics such as communication, cooperation, reasoning, proactive behavior, and adaptivity.

In Section 5, we have shown how each of the three stances can contribute to the use of agents in games. We have also shown some standards and tools that could be used in each of the three stances. We have successfully tested the synchronization principles explained in the infrastructural stance by coupling the Pilgrim game engine (under development at TNO, Soesterberg, The Netherlands) with the 2APL agent platform.

We have shown that the HLA standard is a good starting point to describe the filtering in the conceptual stance. The ease of the translation between the game engine and the agent concepts, of course, also depends on the specific platforms used. The Pilgrim game engine appeared very suitable for this approach because all game objects have a property tree describing all the properties of that object, thus allowing for an easy translation to a common representation (OPM). In a similar fashion, the properties of the 2APL agents are available in a declarative format and could easily be converted to the common representation (OIM). Finally, the agent-based methodology OperA seems to offer a good starting point for combining agent-oriented and game-oriented methodologies.

As a future work, we hope to build some support tools to facilitate the modeling and implementation of games with agents, making use of the framework sketched in this paper.

Acknowledgments

Thanks to Bernard Maassen whose implementation of the connection between Pilgrim and 2APL started off this paper. This research has been supported by the GATE project, funded by the Netherlands Organization for Scientific Research (NWO) and the Netherlands ICT Research and Innovation Authority (ICT Regie).

References

[1] S. Rabin, *AI Game Programming Wisdom 3*, Charles River Media, Brookline, Mass, USA, 2006.

[2] A. Ayesh, J. Stokes, and R. Edwards, “Fuzzy Individual Model (FIM) for realistic crowd simulation: preliminary results,” in *Proceedings of IEEE International Conference on Fuzzy Systems (FUZZ ’07)*, pp. 1–5, London, UK, July 2007.

[3] J. Orkin, “Three states and a plan: the AI of F.E.A.R.,” in *Proceedings of the Game Developers Conference (GDC ’06)*, San Jose, Calif, USA, March 2006.

[4] R. E. Fikes and N. J. Nilsson, “STRIPS: a new approach to the application of theorem proving to problem solving,” *Artificial Intelligence*, vol. 2, no. 3-4, pp. 189–208, 1971.

[5] J. Orkin, “Applying goal-oriented action planning to games,” in *AI Game Programming Wisdom 2*, Charles River Media, Brookline, Mass, USA, 2003.

[6] M. E. Pollack and J. F. Horty, “There’s more to life than making plans: plan management in dynamic, multiagent environments,” *AI Magazine*, vol. 20, no. 4, pp. 71–83, 1999.

[7] M. Lees, B. Logan, and G. K. Theodoropoulos, “Agents, games and HLA,” *Simulation Modelling Practice and Theory*, vol. 14, no. 6, pp. 752–767, 2006.

[8] R. Adobbati, A. N. Marshall, A. Scholer, et al., “Gamebots: a 3D virtual world test-bed for multi-agent research,” in *Proceedings of the 2nd International Workshop on Infrastructure for Agents, MAS, and Scalable MAS*, Montreal, Canada, May 2001.

[9] R. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, Eds., *Multi-Agent Programming: Languages, Platforms and Applications*, International Book Series on Multiagent Systems, Artificial Societies, and Simulated Organizations, Springer, Berlin, Germany, 2005.

[10] A. Newell, *Unified Theories of Cognition*, Harvard University Press, Cambridge, Mass, USA, 1994.

[11] S. Franklin and L. Gasser, "Is it an agent, or just a program?: A taxonomy for autonomous agents," in *Intelligent Agents III. Agent Theories, Architectures, and Languages*, pp. 21–35, Springer, Berlin, Germany, 1997.

[12] "Bos Wars," http://www.boswars.org.

[13] S. Kopp, L. Gesellensetter, N. C. Krämer, and I. Wachsmuth, "A conversational agent as museum guide—design and evaluation of a real-world application," in *Proceedings of the 5th International Working Conference on Intelligent Virtual Agents (IVA '05)*, T. Panayiotopoulos, J. Gratch, R. Aylett, D. Ballin, P. Olivier, and T. Rist, Eds., vol. 3661 of *Lecture Notes in Computer Science*, pp. 329–343, Springer, Kos, Greece, September 2005.

[14] L. Antunes and K. Takadama, Eds., *Multi-Agent-Based Simulation VII*, vol. 4442 of *Lecture Notes in Computer Science*, Springer, Berlin, Germany, 2007.

[15] B. G. Silverman, G. Bharathy, M. Johns, R. J. Eidelson, T. E. Smith, and B. Nye, "Sociocultural games for training and analysis," *IEEE Transactions on Systems, Man and Cybernetics, Part A*, vol. 37, no. 6, pp. 1113–1130, 2007.

[16] L. Padgham, D. Parkes, S. Parsons, and J. Müller, Eds., *Proceedings of the 7th International Conference on Autonomous Agents and Multi Agent Systems (AAMAS '08)*, IFAAMAS, Estoril, Portugal, May 2008.

[17] FIPA, Foundation for Intelligent Physical Agents, http://www.fipa.org.

[18] M. Wooldridge, *Reasoning about Rational Agents*, MIT Press, Cambridge, Mass, USA, 2000.

[19] M. Dastani, "2APL: a practical agent programming language," *Autonomous Agents and Multi-Agent Systems*, vol. 16, no. 3, pp. 214–248, 2008.

[20] R. Bordini, J. Hübner, and M. Wooldridge, *Programming Multi-Agent Systems in AgentSpeak Using Jason*, John Wiley & Sons, New York, NY, USA, 2007.

[21] P. S. Rosenbloom, J. E. Laird, and A. Newell, *The Soar Papers: Readings on Integrated Intelligence*, MIT Press, Cambridge, Mass, USA, 1993.

[22] J. R. Anderson, "ACT: a simple theory of complex cognition," *American Psychologist*, vol. 51, no. 4, pp. 355–365, 1996.

[23] Quake, http://www.idsoftware.com/games/quake/quake3-arena.

[24] Never Winter Nights, http://nwn.bioware.com.

[25] F.E.A.R., http://whatisfear.com.

[26] G. A. Kaminka, M. M. Veloso, S. Schaffer, et al., "GameBots: a flexible test bed for multiagent team research," *Communications of the ACM*, vol. 45, no. 1, pp. 43–45, 2002.

[27] A. Khoo, G. Dunham, N. Trienens, and S. Sood, "Efficient, realistic NPC control systems using behavior-based techniques," in *Proceedings of the AAAI Spring Symposium on Artificial Intelligence and Interactive Entertainment*, Palo Alto, Calif, USA, March 2002.

[28] J. M. P. van Waveren, *The quake III arena bot*, M.S. thesis, Faculty ITS, University of Technology Delft, Delft, The Netherlands, 2003.

[29] A. Witzel and J. Zvesper, "Higher order knowledge in computer games," in *Proceedings of the AISB Symposium on Logic and the Simulation of Interaction and Reasoning*, pp. 1–5, Aberdeen, Scotland, April 2008.

[30] S. Kraus, "Negotiation and cooperation in multi-agent environments," *Artificial Intelligence*, vol. 94, no. 1-2, pp. 79–97, 1997.

[31] T. Mioch, M. Harbers, W. van Doesburg, and K. van den Bosch, "Enhancing human understanding through intelligent explanations," in *Proceedings of the 1st International Workshop on Human Aspects in Ambient Intelligence*, pp. 327–337, Darmstadt, Germany, November 2007.

[32] W. L. Johnson, "Agents that learn to explain themselves," in *Proceedings of the 12th National Conference on Artificial Intelligence*, vol. 2, pp. 1257–1263, Seattle, Wash, USA, July-August 1994.

[33] M. van Lent, W. Fisher, and M. Mancuso, "An explainable artificial intelligence system for small-unit tactical behavior," in *Proceedings of the 19th National Conference on Artificial Intelligence and the 16th Conference on Innovative Applications of Artificial Intelligence (IAAI '04)*, pp. 900–907, AAAI Press, San Jose, Calif, USA, July 2004.

[34] D. Gomboc, S. Solomon, M. G. Core, H. C. Lane, and M. van Lent, "Design recommendations to support automated explanation and tutoring," in *Proceedings of the 14th Conference on Behavior Representation in Modeling and Simulation (BRIMS '05)*, Universal City, Calif, USA, May 2005.

[35] E. Norling and L. Sonenberg, "Creating interactive characters with BDI agents," in *Proceedings of the Australian Workshop on Interactive Entertainment (IE '04)*, pp. 69–76, Sydney, Australia, February 2004.

[36] R. Descartes, *Treatise on Man*, Harvard University Press, Cambridge, Mass, USA, 1976.

[37] IEEE Std 1516-2000, "IEEE Standard for modeling and simulation (M&S) high level architecture (HLA)—framework and rules," September 2000.

[38] Kynogon, http://www.kynogon.com.

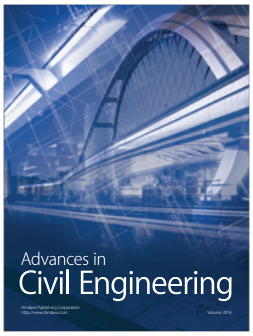
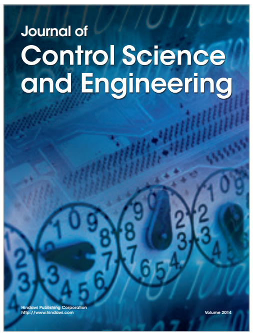
[39] R. Reiter, *Knowledge in Action: Logical Foundations for Describing and Implementing Dynamical Systems*, MIT Press, Cambridge, Mass, USA, 2001.

[40] P. Vorderer and J. Bryant, *Playing Video Games*, Lawrence Erlbaum, Mahwah, NJ, USA, 2006.

[41] K. Salen and E. Zimmerman, *Rules of Play: Game Design Fundamentals*, MIT Press, Cambridge, Mass, USA, 2004.

[42] V. Dignum, *A model for organizational interaction: based on agents, founded in logic*, Ph.D. dissertation, Utrecht University, Utrecht, The Netherlands, 2004.

[43] C. de Melo, R. Prada, G. Raimundo, J. P. Pardal, H. S. Pinto, and A. Paiva, "Mainstream games in the multi-agent classroom," in *Proceedings of IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT '06)*, pp. 757–761, Hong Kong, December 2006.

Hindawi

Submit your manuscripts at
http://www.hindawi.com