SCORE: designing and implementing BDI Agents with the use of interactive computer games as simulation environments

Leonardo Sewald Cunha, Lucia Maria Martins Giraffa
Pontifícia Universidade Católica do Rio Grande do Sul – Programa de Pós-Graduação em Ciência da Computação (PPGCC/PUCRS)
Av. Ipiranga, 6681 – 90610-900 – Porto Alegre – RS – Brazil
Phone: +55 51 3320 3611 – FAX:+55 51 3320 3621
sewald@portoweb.com.br, giraffa@inf.pucrs.br

Abstract
The use of games as testbeds for research projects in the Artificial Intelligence (AI) field is a tradition. Some classic board and card games such as Go, Chess and Checkers were and are extensively used. However, AI researchers are beginning to explore the use of real-time interactive computer games in their research, making this an interesting field to experiment and explore. This paper presents the SCORE project (Simulator for COgnitive agent's behavioR), where we use a Belief-Desire-Intention (BDI) agent specification formalism called X-BDI, applied to a game environment called *Unreal Tournament* (UT). This paper also presents some aspects regarding AI and agent technology applied to interactive computer games, with emphasis in the application of cognitive agent modelling applied to game characters.
Keywords: Artificial Intelligence, BDI Agents, simulation, computer games.

1 Introduction

Interactive computer games have been showing many innovations concerning about performance, interfaces and project techniques. According to [1], the starting point was back at the 70's with the game named *Pong*. It was the first gaming market hit, where two players would compete in a tennis simulation. Since *Pong*, interactive entertainment applications have shown a constant evolution, turning into repositories of technological innovations in different fields, such as Computer Graphics (CG), and AI. Today's modern computer games have state-of-the-art three-dimensional (3D) graphics, many multiplayer features, allowing players to compete against each other or play cooperatively over a network and fairly complex AI implementations. But more importantly, modern computer games are being developed with the use of architectures that allow third-party developers, users and researchers to expand, add or modify game content.

One game that fits into this category is *Unreal Tournament* (UT). Released in 1999 by Epic Games, UT is a First-Person-Shooter (FPS)[1] action game, but with a stronger multiplayer approach than *Unreal* (http://unreal.epicgames.com), its predecessor, both

[1] In the game community, a FPS is a game style where the player visualizes the environment through the character's perspective. As examples we can mention market hits such as *Doom* (http://www.idsoftware.com), *Quake* (http://www.quakeworld.com), *Unreal*, *UT* and others.

using the same game technology and architecture. UT showed many technological advances over its FPS predecessors, not only in CG, but also in the application of AI techniques. Rule-based AI approaches were used in order to improve the playability, aside of the implementation of a "programming technique" called *Extensible AI*, which allows the player to add or modify the game's AI complexity, extending the game's features (these techniques will be described in the next section).

According to [3], real-time interactive computer games offer robust environments for researchers to test and develop AI techniques, aside of the fact that games are relatively cheap and accessible, when compared to industrial or commercial applications. These games have human and/or computer controlled characters populating the game's environment. Since a direct addressing between interactive game characters and agents can be done, as stated by Laird and others in [2], these games become a rich laboratory for AI research, especially in what concerns agent development. These games constitute real products that create real environments with which millions of humans can interact vigorously. For these attributes, the use of real-time interactive computer game environments in intelligent agent research is an interesting field to experiment and explore.

Having this thought in mind, the GAMES AND AI (JEIA) research group at the Computer Science Post-Graduation Program/Pontifícia Universidade Católica do Rio Grande do Sul (PPGCC/PUCRS) developed SCORE, a research project aiming to integrate an agent specification formalism, programming language and tool called X-BDI with the game UT, thus allowing the modelling and implementation of complex BDI agent behaviors in a computer game environment.

This paper is divided in 6 sections. Section 2 presents a description of the main AI and programming techniques used in current computer games, focusing in the agent technology, presented in section 3. Section 4 presents a description of our research work. Final considerations and results achieved with the use of agent technology are presented in section 5. References are presented at the end of this paper.

2 AI and AI-related programming techniques used in games

In [2], we developed a study (survey) about the set of techniques, which represent the state-of-the-art, in what concerns the use of AI and AI-related programming techniques applied to computer games. The main techniques are:

- *Finite State Machines* (FSM's): rule-based approaches such as FSM's and *Fuzzy State Machines* (FuSM's) are the most used AI techniques in computer games, because they are easy to implement and for the fact that they are consolidated as AI tools in the computer game industry. A FSM is made of a set of states, a set of inputs, a set of outputs and a state transition function. This function computes the entries and the current state and outputs a unique new state and a set of outputs. FSM's are usually represented through state transition diagrams, and can be easily implemented through nested *switch-case* commands. For more complex modeling, the FSM's can be built under a graph hierarchy, so that each node in a certain hierarchy level can be expanded to reveal it's dependent

hierarchy, and so on, until the last level, expanding a FSM. These are the *Hierarchical Finite State Machines* (HFSM's), and they provide an efficient way for FSM modular modeling.

The Fuzzy State Machines (FuSM's) are based on Markov chains. Weights are associated to states while transition functions and rules are established to compute the weights of the future states. An example of a game that uses FuSM's is *Call to Power* (http://www.calltopower.com), a strategy game where the player controls civilizations whose profile and characteristical traits were modelled with the use of FuSM's. The use of FSMs allows the user to build elements with a relatively complex behavior;

- *Extensible AI*: this is not exactly an AI technique; it's more likely to be considered as a programming technique. Through this programming style, the player has the possibility to create character behaviors or modify pre-existent character behavior. *Extensible AI* usually appears in the form of *script* languages, that the game player/programmer can use to extend the game's features. The game developers must build a *script* compiler which is inserted in the game's executable file, or as a separate program shipped with the game). The *scripts* are based on function calls to the game's internal AI subsystem, which cannot be modified by a regular *Extensible AI* implementation. Many games shipped in the last few years implement this technique, following a trend that started with games such as *Duke Nuke'em 3D* (http://www.dukeworld.com) and *Quake*;
- Search algorithms – the *A-Star* (A*): According to [17], the A* is the most used search algorithm in computer game design and implementation. The A* uses a heuristic function which determines the quality of each of the possible states (nodes). The function estimates the cost of the paths towards the destination, passing through the current node, and it chooses the best way to achieve it. The quality of the node is measured by lowest cost among all candidate nodes. The performance of its implementation depends on the heuristics enclosed in the heuristic function. A bad heuristic function can drastically reduce the algorithm's speed, or produce incorrect routes. In order to achieve the best results, the heuristic function must be admissible. It means that it must be an underestimate of the actual cost of moving from the current node to the goal node. Usually, game developers are very familiar with search algorithms. Thus, now they are focusing on the association of search techniques with specific situations, such as in pathfinding associated with *terrain analysis*[2], a situation usually found in strategy games such as *Age of Empires II: The Age of Kings* (http://www.ensemblestudios.com/aoeii/index.shtml), a game where the player must develop ancient civilizations;
- Neuronal Networks and Genetic Algorithms: According to [18], Neuronal Networks (NN's) use a massive interconnection of computational cells called

[2] The *terrain analysis* is a programming technique that is used to point out locations on the game maps which can be difficult for the game elements to move through, such as bridges or mountain passes. A good *terrain analysis* can output valuable information for the game's pathfinding system so that the latter can solve more complex search problems.

"neurons" or "processing units". The links between these neurons are called synapses. Each neuron has an associated value (weight), which is used to store acquired knowledge. The neuronal net's algorithm can learn and keep the changes of these weights in order to improve the neuronal network. A neuronal net's learning process is called supervised if the expected output is known. Otherwise the learning process is called unsupervised. Genetic algorithms (GA's) are computational models inspired on human evolution. They typically represent knowledge through binary or Boolean attributes. According to [19], a regular GA implementation starts with a set population with some attributes (chromossomes). These structures are evaluated by a fitness function and reproduction opportunities are offered. The set of attributes, which represents a better solution to the problem, has a better chance to reproduce. This evolution process can be associated to the learning process for GA's. There are some game-related projects that explored a combination of GA's and NN's, and one of the most significant is the *NeuralBot* (http://www.botepidemic.com/neuralbot), a computer-controlled opponent (*bot*) created to be used in the game *Quake II* (http://www.quakeworld.com). The *bot* uses a neuronal net to control its actions and a GA to train its neuronal net. By this way the *bot* does not need any sort of pre-programmed behavior: it can learn and adapt itself using the environment inputs;

- Artificial Life (A-Life): according to [17], the *A-Life* representation techniques provide flexible ways to create realistic behavior in game characters. *A-Life* aims to simulate the behavior of real world living organisms through different methods, such as rule-based approaches, GA, among others. Instead of implementing a variety of complex behaviors, the *A-Life* approach divides these complex behaviors in small parts, in order to simulate simpler behaviors. A decision-taking hierarchy, used by the game elements to decide what actions must be taken interconnects these small parts. Combinations of low-level behavior sequences can automatically generate higher-level, complex behaviors, thus reducing implementation complexity. *A-Life* techniques are usually used in simulators. However, some action and strategy games, such as *Unreal* and *Age of Empires II,* already use *A-Life* approaches to control group movement, such as *flocking algorithms*, where the *flocking* techniques are used to control the coordinated movement of groups of fishes and birds, and to control army formations;
- *Software Development Kits* (SDK's): SDK's, or simply *toolkits*, implement different sets of AI techniques, making them ready to be used by software developers at a higher abstraction level. For this attribute SDK's also aren't AI techniques, but are considered as AI-related programming tools. They can speed up the application development process, since developers just have to use the functions implemented in the toolkit, instead of implementing their functionality (for example, when using a FSM approach. The developers can use a ready-made toolkit implementation of general FSM functionality and tune it up for their project). Some SDK's were created specifically for use in game projects,

while others have a wider usage scope. Some examples of SDK's are: *Motivate* (http://www.motion-factory.com), *Spark!* (http://www.louderthanabomb.com) and *DirectIA* (http://www.animaths.com).

3 Computer Games implemented with the use of Agent Technology

According to [4], an agent is a computer system that is capable of independent action on behalf of its user or owner. A Multiagent[3] system is one that consists of a number of agents, which interact with one another. In order to successfully interact, agents must have the ability to cooperate, coordinate, and negotiate. According to [20] and [4] the agents have different characteristic attributes (properties). However, some properties are necessary to be observed when building an agent:

- Autonomy: refers to the fact that agents can operate in an independent way, without human supervision. Agents usually implement this property in a certain degree;
- Reactivity: refers to the agent's ability to react to environment inputs. A reactive system is one that maintains an ongoing interaction with its environment, and responds to changes that occur in it (in a reliable time of response);
- Social ability (interaction): refers to the interaction with other agents (and possibly humans) via some kind of agent-communication language.

We adopted the definition used by [5], which defines an agent as an entity, which has goals, that has the capacity of perceiving certain properties in the environment. They can sense the surrounds and can act in this environment, and some of these actions and/or perceptions can be done through communicating with other agents.

According to many authors mentioned by [6], there is a taxonomy[4] being widely used in the Distributed Artificial Intelligence (DAI) community, which classifies agents in two groups: reactive and cognitive agents. Cognitive agent architectures are typically unable to act fast and adequately in unpredicted situations, while reactive agents are unable to discover alternatives to their behavior when the environment status is too different from their initial goals, making them less flexible. Table 1 presents some comparisons between reactive and cognitive agents in what concerns their basic features.

Table 1: Reactive Agents X Cognitive Agents

Reactive Agents (non-deliberative)	Cognitive Agents (deliberative)
Reactive agents do not have an internal symbolic environment representation	They have goals and an internal world representation, which contains information about application requirements and action consequences that can be explicitly represented
Action selection is directly associated with the occurrence of a set of events in the environment	Action selection is done through an explicit deliberation over different options. For example, by using an internal world representation, a plan or

[3] Multiagent systems provide a new tool for simulating societies, which may help shed some light on the various kinds of social processes. Thus, agents can be considered as a tool for understanding human societies, or other societies based on the human ones.

[4] This taxonomy has more of a didactical application than real/practical applicability.

	even a function which evaluates an action according to its use
Architectures have a fixed, rule-based action control mechanism	They have an internal planning system based on successive refinements, using the information provided by the world representation to build a plan which can reach the agent's goals
Demonstrate excellent performance in real-time environments, although there is the need of a previous effort to determine specific solutions to the possible situations	Lack of speed in real time environments. The amount of time needed to analyze the situations, to build and to reuse plans, typically can make the agent very slow agent to act in a real time environment
In general they don't have explicit goals which can be arbitrated and altered during the execution of the system	Given a certain goal and the knowledge that a certain action will lead it to this particular goal, then the agent selects this action
Built with the use of simple control mechanisms such as FSM's or rule-based approaches	Originated from the classic planning systems, which output a sequence of correct actions (plans) to reach a certain objective

An example of a game that uses reactive agents is *Guimo*, a game released in 1997 by Southlogic. The agents were built using the HFSM approach. The HFSM has states such as *Wander*, *Fallback* and *Attack*, and each one of these "macro-states" has an internal FSM, which executes the specified actions. In contrast, an example of a cognitive agent approach in games is shown in *Black and White*, released by LionHead in 2001. Agents were modeled based on a BDI architecture. The agent receives an "inventory" of locations of certain points. Of those descriptions, the agent then creates "opinions" over which types of objects are more appropriate to satisfy its goals.

In order to solve the main restrictions of these two architectures, the hybrid architectures were adopted. Hybrid architectures combine reactive and cognitive techniques. According to [5], hybrid agents use an off-line planning system (a planning system which is not executed in real time), for the generation of plans in a higher abstraction level, while decisions are made on the lower level. Refinement alternatives for plan steps are handled by reactive systems.

An example of a hybrid agent application is the *Multi-Cooperative Environment* - MCOE, a multimedia educational game, created as a prototype for Giraffa's PhD thesis [7]. The thesis work proposed to model an Intelligent Tutoring System (ITS) through the use of agent technology using a Multiagent System architecture.

4 Integrating Computer Games and Intelligent Agent Research

This section presents the SCORE system, the research project developed in the JEIA group. The subsections below present a description of the system, alongside with implementation details.

4.1 Description

Some modern computer games have their *engines*[5] implemented as *Dynamic Link Libraries* (DLL's). They are files that store the system's object/function libraries. These DLL's are used to manipulate game data (textures, graphical elements) and interact with certain parts of the system, which typically are external to the engine itself, such as music/sound effects, AI and other subsystems.

Through the use of some tools (in most cases, shipped with the game) these subsystems' source code can be viewed and manipulated, allowing the user to change many game attributes without the necessity of acquiring the game engine's source code. These subsystems can be considered as the engine's programmable interfaces (since the game engine interprets the code for these subsystems), as they allow access to internal game functions. Most of these interfaces are accessible with the use of script languages, whose commands and reserved words assigned to game AI tasks are usually associated with low-level actions. The developer and/or researcher could directly program a character's behavior by using these interfaces. The game UT fits in this category.

The SCORE project uses the game UT as a testbed. The game's architecture approach consists of the use of a Virtual Machine, a Compiler and script and byte code, similar to the Java language architecture, as shown in Figure 1. The *Init* sequence compiles the script code into byte code and loads it on the Virtual Machine to be executed. From this point, the game's execution loop begins. During the game's execution, the system's state changes. The Virtual Machine executes functions and service calls until an "EndGame" request is called.

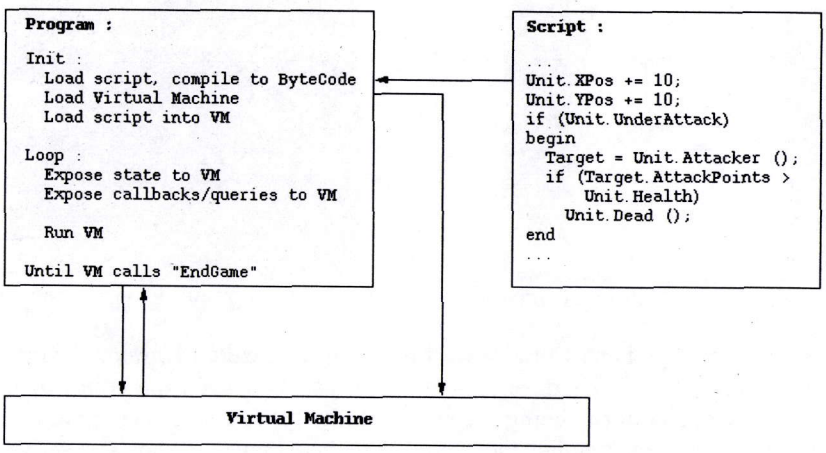

Figure 1: UT architecture scheme

[5] In what concerns computer game development, an *engine* consists of a set of methods/functions and data structures created to ease the manipulation of game data, thus assisting programmers in creating computer games. The engine is tipically considered as a game's core.

UT has a built-in, integrated script language, called *UnrealScript* (US). The script language was created to provide a personalized game programming language, which would provide native support for features such as states, time and network communication. UT's development team first used US during the game's implementation phase. Now many UT players who have programming knowledge use the script language to extend the game's features.

The language is object-oriented, mixing C++ and Java features. The implemented code is compiled by using the UCC tool, a compiler created by UT's development team to be used with the language, and then be executed by the *Unreal Virtual Machine*, which is basically represented by the game's engine. For more information on the US language, the work of [8] can be consulted.

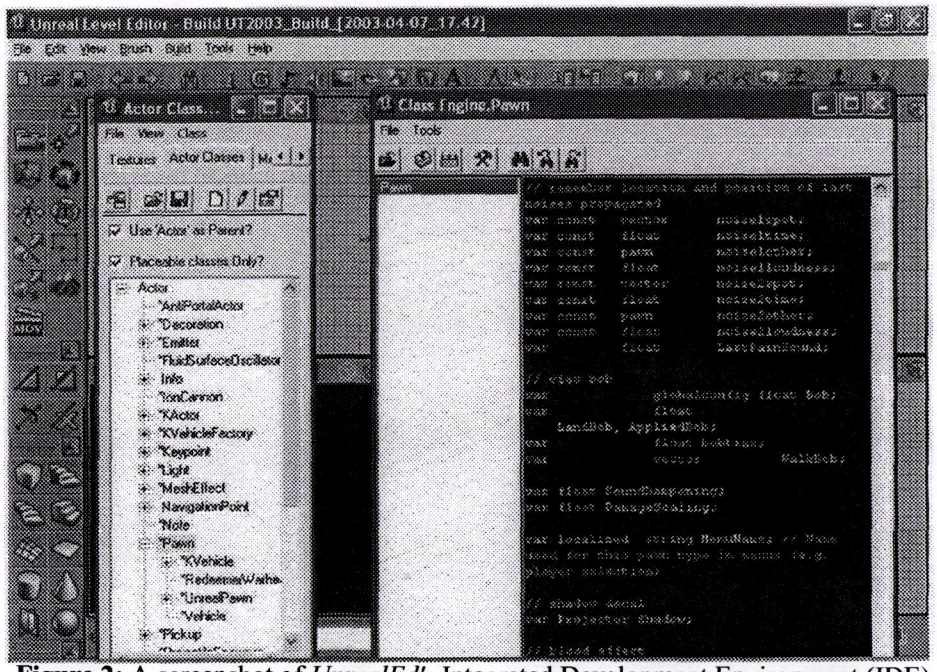

Figure 2: A screenshot of *UnrealEd's* Integrated Development Environment (IDE)

The script language is part of a powerful editing tool called *UnrealEd*, and can also be accessed through it. UT's development team also first used the editor to build the game, and now this editor is being shipped with the game itself, accessible to all users who wish to change/manipulate the game elements. Figure 2 shows a screenshot of *UnrealEd*. All the game elements (music, sound effects, textures, characters, and weapons, enemy AI) can be manipulated through the editor and/or *UnrealScript*. The screenshot in Figure 2 shows two script-related windows: the one on the left shows a part of the game's object hierarchy, while the one on the right shows the *UnrealScript* code for the class *pawn*, which represents a game element which has AI capabilities.

The user can change the code or inherit the properties of other superclasses and then recompile the game's script code[6], this way changing the game.

UT and UnrealScript were chosen for use in this project because there is free on-line available documentation, aside of our interaction with the development team members. Also, the game allows free access to most part of the game's code through US and *UnrealEd* tools.

Through the use of US (which can be considered as UT's programmable interface), it would be possible to manipulate character behavior. However, it would be quite a hard task to create complex (intelligent) cognitive agent behavior by direct script language manipulation. Fortunately, UT's game interface can be controlled via external software.

At this point, there is the necessity to use tools in order to reduce agent modelling and implementation complexity. Our research group has two available tools to attend this issue:

- X-BDI: X-BDI is an agent development and testing tool based on the now common concept of beliefs, desires and intentions [9]. Derived from a formal model of BDI agents [10], X-BDI implements the algorithms that deal with the interaction of mental states, namely: how to keep beliefs consistent, how to keep intentions consistent (internally, and with beliefs), how to derive intentions from desires and plans from intentions. Therefore, X-BDI provides a tool that, when fed with the description of an agent in terms of its beliefs and desires and given input from the environment, is capable to manage these mental states and produce sequences of actions which satisfy the agent's desires, and passes them to the environment. X-BDI is not seen as a complete agent but as a cognitive kernel that is to be part of an agent;
- E-BDI: the E-BDI system is a BDI agent programming editor based on the X-BDI environment, making it more operational and usable. The editor allows the developer to define the set of BDI mental states and their interrelations by using both a text and graphic-oriented interface [11]. In this way, the E-BDI editor addresses one of the main issues in BDI agent programming the complexity to visualize and debug large BDI mental state descriptions. The developer can analyse the defined mental states, making it easier to detect the presence contradictions or deadlocks [11].

We are using some previous results reached by [10], [7], [6], [11] and [12] to control character behavior in interactive computer games. We are using the mental state approach in a similar way (although with a different tool, the X-BDI environment) as in Laird's *Human-AI* project [3], where the Soar[7] architecture is used to control characters in dynamic environments.

---

[6] It is important to state that by using the editor, the user cannot change the game's engine. In fact, the editor uses the engine's resources to allow the user to manipulate the game elements.

[7] According to [Laird 2001], Soar is an unified architecture for the development of systems which show intelligent behavior. Built based on theories about the human problem-solving abilities, the Soar architecture provides fixed computational structures, over which knowledge can be coded and used in order to produce actions in search for goals (a deliberative behavior).
Among the projects where the Soar architecture was used, let us highlight the research project which served as a motivation for our work: the *human-level AI* research project, coordenated by Laird [2001], where the Soar architecture was integrated with the game *Quake II*.

The X-BDI system has a higher abstraction level than the game's built-in *script* language, making it an essential tool for cognitive agent modelling. Thus, it is necessary to build an intermediate layer to establish the communication between the game's programmable interface and the agent's cognitive kernel, as depicted in Figure 3.

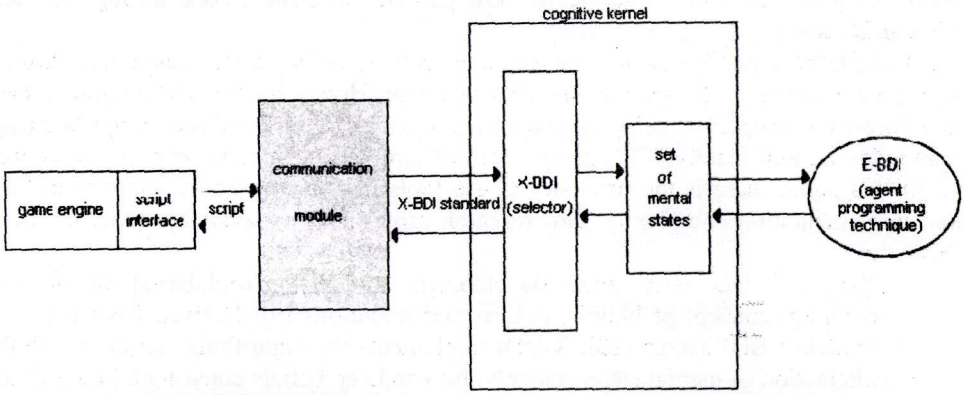

Figure 3: Controlling game characters of a computer game via X-BDI

In order to establish such a communication level, a two-way translator was defined. In the game -> X-BDI way, the translator maps game outputs, coded in *US*, as inputs representing the agent's environment sensoring, coded in the BDI format used by X-BDI. This information is then processed by X-BDI. The vice-versa process, translating X-BDI-generated plans into action sequences, coded in *US*, to be executed by the game characters, is also made using this translator.

Table 2: Code examples from US and X-BDI

US code	X-BDI code

Else if (Orders == 'Patroling')
GotoState('Patroling');
else if (Orders == 'Attacking')
GotoState('Attacking');
else if (Orders == 'Ambushing')
GotoState('Ambushing','FindAmbushSpot');
else if ((LikelyState != '') && (FRand()< 0.35))
GotoState(LikelyState, LikelyLabel);
Else StartRoaming();...
state Attacking{
ignores SeePlayer, HearNoise, Bump, HitWall;
function ChooseAttackMode() {
local eAttitude AttitudeToEnemy; local float Aggression; local pawn changeEn; local TeamGamePlus TG; local bool bWillHunt; bWillHunt = bMustHunt; bMustHunt = false;
if ((Enemy == None)||(Enemy.Health
	Identity(sewald).
	des(sewald,MeleeAttack(Enemy)) if
	bel(sewald,BumpInEnemy(OK)),
	bel(sewald,ReadyToAttack(OK)),
	bel(sewald, EnemyDistance(OK)),
	bel(sewald, KeepOnMelee(OK)).
	act(sewald,DetectPawn(Enemy)) causes
	bel(sewald,BumpInEnemy(OK)) if
	
	bel(sewald,ThereIsEnemy(OK),t1).
	act(sewald,IsReadyToAttack(ENEMY)) causes
	bel(sewald,ReadyToAttack(OK)).
	act(sewald,GoinToState(TAKEHIT)) causes
	bel(sewald,KeepOnMelee(OK)).
	act(sewald,DistanceTest(OK)) causes
	bel(sewald, EnemyDistance(OK)).

<= 0)) { WhatToDoNext(","); return; }	

Table 2 shows code samples from UT's script code (taken from the *ScriptedPawn* character class), and X-BDI agent code. The X-BDI language is similar to PROLOG, having low complexity in what concerns it's syntax. The *US* language has a native implementation of states, used mainly for AI purposes, reducing the complexity to program character behavior. The state concept implemented in *US* defines sequences of actions, which can be associated to the actions performed by a X-BDI agent. The conditions that determine if an agent will or will not enter a given state are associated with beliefs of a X-BDI agent.

The US language supports state machine programming at the language level, through the use of the *state* command. With the use of this command, the programmer can directly define the character's states (e.g. attacking, dying, wandering, patrolling, meleeattack), where the character performs certain actions. The state transition is done with the use of the *GotoState(next_state)* command, where *next_state* represents the character's next state.

In order to test the translator's functionality, a prototype was built. A new character class was created, extending the *Brute*[8] character class. From the new class, called *Sewald*, there was selected the set of behaviors which implement the *MeleeAttack* state, where the enemy character engages in close combat. The new class was created because there was the need to implement a way to force the UT system to write game execution information to its output log files.

4.2 Implementation

The translator has two basic functions:

- Convert the character's conditions to enter a given state into BDI agent beliefs;
- Convert BDI agent's desires into game's state transitions.

In the first case, the translator follows an association table, listed in Table 3. For the second translator feature, the translation process is straightforward: the BDI agent's desires are directly associated with the game's internal state transition command, *GotoState()*, with the need of syntax modifications.

Table 3: US to XBDI code translation table for the *MeleeAttack* state

Situation	X-BDI code	US code
Character touches another game object, and the touched object is another enemy, or the player	BumpInEnemy(OK)	Enemy!=None; Other==Enemy; Pawn(Other)!=None; SetEnemy(Pawn(Other));
Character is ready to attack	ReadyToAttack(OK)	bReadyToAttack==True;
Character is at an acceptable range from the victim, in order to engage in close-combat	EnemyDistance(OK)	VSize(Enemy.Location - Location) <= MeleeRange + Enemy.CollisionRadius + CollisionRadius;

[8] Class that implements a specific computer-controlled enemy, with it's own AI functionalities, graphical textures and associated sound events.

Others: character is going to the *TakeHit* state; character is at the *MeleeAttack* state itself, in the *AnimEnd* method	KeepOnMelee(OK)	NextState == TakeHit;

The translator was implemented as a black box module, where the results processed by the X-BDI environment are the module's input parameters, while the module outputs the corresponding US commands. The same module also does the vice-versa process, but with the use of a different function.

The execution of the system as a whole begins when the translator starts searching in the UnrealTournament.log (UT's log file) file for the string "SCORE FOUND#. This string is a flag, which indicates that the game character is about to enter in a given state (meaning that the game character's AI script execution has evaluated all the conditions ("*ifs*") to enter the state). For example, the set of conditions which represent the *BumpInEnemy(OK)* BDI agent belief (listed in Table 3), is written in UT's log file as "SCORE FOUND#BumpInEnemy".

The translator then processes the flag, converting it to a X-BDI command. The translated command is passed to X-BDI through its input agent coreography file, named COREO.A. Following the above example, the flag is converted to the *DetectPawn(Enemy)* X-BDI command, indicating that the game character has satisfied all the necessary conditions, so that the X-BDI agent assumes *BumpInEnemy(OK)* as a belief. Figure 4 shows the code excerpt that handles the above situation. Variables *fpUT* and *fpOut* represent, respectively, the UnrealTournament.log and COREO.A files.

```
while(fgets(cBuf, 100, fpUT))
  if(cPtr=strstr(cBuf, "SCORE FOUND#"))
  //moves the pointer
  cBuf[strlen(cBuf)-1]='\0';
  cPtr+=(char)strlen("SCORE FOUND#");
  //BumpInEnemy
  if(!strcmp(cPtr, "BumpInEnemy"))
    fputs("DetectPawn(ENEMY)\n", fpOut);
  //ReadyToAttack
  if(!strcmp(cPtr, "ReadyToAttack"))
    fputs("IsReadyToAttack(OK)\n", fpOut);
  //EnemyDistance
  if(!strcmp(cPtr, "EnemyDistance"))
    fputs("DistanceTest(OK)\n", fpOut);
  //KeepOnMelee
  if(!strcmp(cPtr, "KeepOnMelee"))
    fputs("GoinToState(TAKEHIT)\n", fpOut);
```

Figure 4: Code excerpt for UT's log file reading and conversion

The BDI agent then interprets, through it's source code (shown in Table 2), the command received through COREO.A. After interpretation, the agent sends out a string containing a desire (the interpretation results). This string is also a translator flag, this time indicated by the "SCORE SAYS#" string. The desire is read by the translator, which converts it to a *GotoState()* command. Figure 5 shows the code excerpt that implements this situation.

```
while(fgets(cBuf, 100, fpOut))
if(cPtr=strstr(cBuf, "SCORE SAYS#"))
//move o ponteiro
cBuf[strlen(cBuf)-1]='\0';
cPtr+=(char)strlen("SCORE SAYS#");
//MeleeAttack
if(!strcmp(cPtr, "MeleeAttack"))
fputs("GotoState("MeleeAttack")\n", fpUT);
```

Figure 5: Code excerpt which converts XBDI output to US code

The black box module receives two command-line parameters: the conversion type (UT to X-BDI or X-BDI to UT) and the input and output filenames. The conversion type identifiers are:

- -ux: converts from UT to X-BDI;
- -xu: converts from X-BDI to UT.

Two separate conversion functions were implemented:

- int UT_XBDI(char *input_ut, char *output_xbdi);
- int XBDI_UT(char *input_xbdi, char *output_ut).

The UT_XBDI function (used when the *-ux* parameter is passed) searches for the flags written inside UT's log file (passed as the input file) and outputs the converted X-BDI code in the informed output file, following the associations listed in Table 3.

The XBDI_UT function (used when the *-xu* parameter is passed) reads the X-BDI generated output file, passed as the translator's input file, converting the desires into US state transitions.

Table 4 summarizes the use of both functions.

Table 4: Description of the implemented functions

Function name	Description
UT_XBDI	- reads from UT's log file; - for each line containing the "SCORE FOUND#" flag, read the rest of the line and convert to the correspondent X-BDI command (X-BDI agent's environment sensing); - writes to output file, to be read by X-BDI;
XBDI_UT	- reads from X-BDI generated output file; - for each line containing the "SCORE SAYS#" string, read the rest of the line and convert to US commands, representing a state transition; - writes to the output file, to be read by UT;

5 Conclusion

The AI researchers have been developing many works using classic games such as chess, checkers or puzzles, and so on. However, modern computer games are, in their majority, played in real time, in virtual environments, which involve a high level of dynamism and interactivity. Thus, real time computer games are an interesting topic for study and implementation of AI techniques. There is also the fact that there are very few

research projects involving modern computer games in the academic community. There is some kind of misunderstanding or resistance over the use of interactive computer games as powerful testbeds for AI techniques. The works [3] and [13] have developed with the use of real-time computer games are good examples about the potential of such applications.

According to [3], the main reason that makes the academic community ignore real-time interactive computer games is associated to the fact that, usually, the objective of these game's AI systems is not to create intelligent characters, but to improve the game's playability level through the illusion of intelligent behavior. Another reason to justify the shortage of scientific material on the subject would be the fact that many game developers like to "reinvent the wheel", creating their own game development methodologies instead of using ready-made ones.

The JEIA research group from PUCRS (website at: http://www.inf.pucrs.br/~giraffa/jeia/index2.htm) is trying to contribute to modify this unfavorable scenario. Research with the use of interactive computer games opens a new door in real-time simulation. Since some modern games have expandable and open-source architectures, the list of application fields can be wide. For instance, in the case of our project, the SCORE system, upon its completion, can be used (alongside with UT's development tools) as a higher-level BDI agent development and testing tool, encapsulating the X-BDI formalism as its development tool, the translator as its conversion grammar and UT as its game environment testbed, associated with a multi-purpose environment generator, here represented by UT's *UnrealEd*.

As examples of the research being developed by the group, we can mention: MCOE [7]; RL_MCOE, proposed by [14], is an application of reinforcement learning techniques using the work implemented by [7]; TCHE by [15], and QUERO-QUERO by [16] are educational games created to assist children to develop basic Math concepts. These game's environments were developed by multidisciplinar teams and tested in real classroom environments. REVOLUTION by [21] involves the use of Role-Playing Games (RPG's) in the teaching process. The JEIA group has been developing research involving agent technology applied to educational game modelling and implementation. The group is now starting to develop applications involving real-time interactive computer games. More detailed information is available at http://www.inf.pucrs.br/~giraffa.

Our work described in this paper also intends to demonstrate the potentiality of BDI agent techniques for game projects. Moreover, considering anticipation as a form of planning, XBDI is capable of constructing anticipatory agents that, by using the SCORE system, can then be used as in-game agents. As important contributions from this work, we can mention:

- Contribute to spread out interactive computer game research in the academic research community;
- Provide tools to develop an application which will allow behavior modelling and programming through E-BDI's associated programming technique;
- Contribute with BDI agent research.

As suggestions of future work, we can mention:

- Provide a standard for X-BDI output manipulation: by having a standard way to manipulate X-BDI output, we can expand the communication layer in order to enclose different languages, enlarging it's usage scope;
- Implement a more dynamic communication interface: in the project's current state, it is possible to model and program high-level behavior for BDI agents, and visualize the results in an interactive game environment. However, the interaction between the game and the X-BDI environment is not fully dynamic (real-time), due to project time restrictions. Further analysis of the game's script code is needed to implement a more dynamic solution, in order to make the system more usable.

Acknowledgements

The group would like to thank Dell Computers and Microsoft for supporting this research project.

References

[1] BATTAIOLA, A. L. Jogos por Computador – Histórico, Relevância Tecnológica e Mercadológica, Tendências e Técnicas de Implementação. XIX Jornada de Atualização em Informática, 2000, ed. Universitária Champagnat, pp. 83-122.

[2] CUNHA, L. S and GIRAFFA, L. M. M. Um estudo sobre o uso de Agentes em Jogos Computadorizados Interativos. PPGCC/PUCRS, Porto Alegre, 2001. Technical Report (available at http://www.inf.pucrs.br/ppgcc/).

[3] LAIRD, J. E. Using a Computer Game to Develop Advanced AI. Computer, 34, (7), 2001.

[4] WOOLDRIDGE, M. An Introduction to Multiagent Systems. England: John Willey & Sons, 2002.

[5] NAREYEK, A. Intelligent Agents for Computer Games. Captured on April 2001. *Online.* Available on the Internet at: http://www.ai-center.com/references/nareyek-00-gameagents.html

[6] VICARI, R. M and GIRAFFA, L. M. M. The Use of Multi Agent Systems to Build Intelligent Tutoring Systems In: International Journal of Computing Anticipatory Systems. The American Institute of Physics (AIP), 2002.

[7] GIRAFFA, L. M. M. and VICCARI, R. M. Estratégias de Ensino em Sistemas Tutores Inteligentes modelados através da tecnologia de agentes. Revista de IE/SBC. (6), 2, 1999. (Brazilian Journal of Computer Science applied to Education)

[8] CUNHA, L. S. and GIRAFFA, L. M. M. UnrealScript Language Syntax. PPGCC/PUCRS, Porto Alegre, 2001. Technical Report (available at http://www.inf.pucrs.br/ppgcc/).

[9] WOOLDRIDGE, M. Reasoning about Rational Agents. Massachusetts: The MIT Press, 2000.

[10] MÓRA, M.C., LOPES, J.G., COELHO, J.G. and VICARI, R. BDI models and systems: Reducing the gap. In: Agents Theory, Architecture and Languages Workshop. Lecture Notes on Artificial Intelligence, Springer-Verlag, 1998.

[11] ZAMBERLAM, A. O. et al. E-BDI: um editor para programação orientada a agentes BDI. In: III ENIA Encontro Nacional de Inteligência Artificial, 2001, Fortaleza. XXI Congresso da Sociedade Brasileira de Computação. Fortaleza. Proceedings: SBC, 2001

[12] GOULART, R. R. V. et al. Auxiliando o tutor na gerência das informações do ambiente e dos alunos. XI SIMPÓSIO BRASILEIRO DE INFORMÁTICA NA EDUCAÇÃO, 2000, Maceió, Alagoas. Proceedings: SBC, 2000.

[13] ADOBBATI, R. et al. GameBots: A 3D Virtual World Test-bed for Multi-Agent Research. Proceedings of the 2nd. Workshop on Infrastructure for Agents, MAS and Scalable MAS, ACM Press, New York, 2001, pp. 47-52.

[14] CALLEGARI, D. A. Aplicando aprendizagem por reforço a uma arquitetura multiagente para suporte ao ensino de educação ambiental. PPGCC/PUCRS, Porto Alegre, 2000. Masters Thesis.

[15] MAZZORANI, A. C. et al. Tchê: Uma Viagem pelo Rio Grande do Sul. FACIN/PUCRS, Porto Alegre, 2001. Trabalho de Conclusão II.

[16] COMUNELLO, G. et al. Quero-Quero Aprender Matemática. PPGCC/PUCRS, Porto Alegre, 2001. Trabalho de Conclusão.

[17] WOODCOCK, S. Game AI: The State of the Industry. Captured on April 2001. *Online*. Available on the Internet at: http://www.gamasutra.com/features/20001101/woodcock_01.htm

[18] HAYKING, S. Redes Neurais: Princípios e Prática. Bookman, 2001.

[19] WHITLEY, Darrel. A Genetic Algorithm Tutorial. Statistics & Computing. (4), ,1994.

[20] RUSSEL, S.; NORVIG, P. Artificial Intelligence: A modern Approach. Prentice-Hall, 1996.

[21] Bittencourt, J. R.; Giraffa, L. M. M., A Utilização dos Role-Playing Games Digitais no Processo de Ensino-Aprendizagem. PPGCC/PUCRS, Porto Alegre, 2003. Technical Report (available at http://www.inf.pucrs.br/ppgcc/).