TOWARDS INTERFACING BDI WITH 3D GRAPHICS ENGINES

N. P. Davies, Q. H. Mehdi and N. E. Gough

Games Simulation and AI Centre
Research Institute for Advanced Technologies
School of Computing and Information Technology
University of Wolverhampton
Wolverhampton, UK
E-mail: N.P.Davies2@wlv.ac.uk

KEYWORDS
artificial intelligence, BDI agents, game engines, graphic engines, GameBots

ABSTRACT

This paper presents work in progress towards the goal of creating human-like artificial intelligence that interfaces with a 3D virtual environment to control computer-generated characters. We will outline out current development regarding the creation of BDI agents using the AI tool JACK, and how we intend to create a link between JACK and sophisticated graphics and game engines including Irrlicht and Unreal Tournament. We will also outline future aims including the introduction of a messaging protocol that will be used for AI/game communication. Using the techniques we hope to achieve behaviour in computer game characters that will appear more realistic than current reactive techniques.

INTRODUCTION

The AI architecture we are implementing is based on a simplified view of human cognition known as Belief-Desire-Intention (BDI) as postulated by Bratman (1987). The architecture is capable of modelling expert human behaviour, and has been used in applications such as Air Traffic Control (Rao & Geofgeff,. 1995). The architecture is gaining interest from the computer game research community, with some success (Norling. 2004). There are still many unexplored research areas, including the role of memory and emotion on decision-making processes, the effect of physical conditions of an agent e.g. fatigue, and the aspect of team coordination with social hierarchies where tasks need to be distributed.

We outline here our progress towards the creation of BDI agents, and identify how we intend to interface the intelligence with a sophisticated graphics engine. We have identified a set of potential tools for the creation of our AI, and implemented some agents capable of performing basic behaviour using the commercial BDI platform JACK. The agents take the form of fish in a fish tank that swim until they get hungry. Once hungry they adopt plans to find food. Currently the agents are visualised in a 2D java environment. As this environment is quite simplistic, the next stage of our development will involve creating more complex agents and situating them in more advanced 3D environments allowing for more sophisticated scenarios to be experimented with such as those found in modern computer games and simulation. We outline here several ways we can accomplish this including the use of commercial game engines, the creation of a custom game engine using a platform API such as DirectX, or the use of commercial and open source graphic engines. We will also outline some of the issues involved in linking a Java based AI system with a game engine via a messaging protocol.

EXAMPLE BDI APPLICATION

We have developed some simple AI agents that have the ability to respond to environmental events, identify appropriate plans to handle the events, and execute those plans in a timely manner. While executing plans, the agent monitors the environment and an internal belief structure in order to ensure the plan is still relevant. The AI architecture we are using is BDI as it exhibits many of the features we require. In this architecture, an agent is characterised by its beliefs about the state of its environment, goals (desires) that it wishes to achieve in this environment, and a set of plans and partial plans it can use in order to satisfy its desires. This architecture is outlined more thoroughly in Davies et al (2005). Implementing a sufficiently sophisticated BDI system from scratch is a significant development problem. As an alternative we have chosen a BDI tool that incorporates the features we need. Many of these tools exist including JADE (TiLab 2005) with the BDI extension JADEX (Braubach & Pokahr. 2005), and JASON, which is an interpreter for the agent oriented BDI programming language AgentSpeak. However, we have chosen to implement our system using the commercial platform JACK (Agent Oriented Software 2005), as it appears to be the most advanced tool available, and has interest from the computer game research community (Norling 2004).

Our first application scenario is based in a 2D fish tank environment (figure 1). Fish swim around the environment using up their energy reserves. Once their energy reaches a sufficiently low level they become hungry. To replenish their energy they can eat food available in the environment. The agent designs are quite simple, but exhibit some of the fundamental properties we will develop in future work. Agents comprises of plans, beliefs and events (Figure 2). During execution, fish agents are created that comprise of some internal data including position variables (X and Y), a target position (X and Y) that the fish are trying to get to, and a health value. While the fish are moving the health values diminish. This internal data can be considered the agents belief structure. Food agents are also created that contain position information, and a value for the health they can supply once eaten. The next element of the agent design is to specify the events and plans the agents can handle. Plans will be outlined in more detail later; however, an

example of a plan developed in our application is 'Find Food Plan' that sets the target location of the fish to the nearest food source. This plan is triggered when an event messages is sent to the agent.

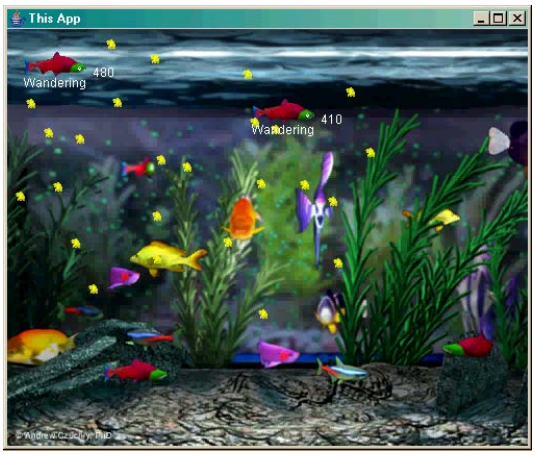

Figure 1: Simple BDI agents

There are two types of events; internal and external. External events are triggered by the environment and other agents via messages. Internal events are generated when certain conditions are reached in the agent's data structure. For example, internal events created by fish agents include 'Find Food Event'. This is triggered when health falls below a certain trigger level. Another example is 'Find New Target Event' that occurs when the agent has reached a goal position. External events consist of 'Food Found Event' that is triggered by the environment when an agent reaches a food source. The 'Food Found Event' is raised by a collision detection algorithm that checks fish agent position against food agent positions. Both internal and external events are dealt with via a corresponding plan that specifies an appropriate course of action to take. To determine which plans are appropriate to an event a hierarchical plan selection system is enforced. Each plan can have its relevancy specified in relation to a particular event. That is, a plan has a list of events it is designed to handle. Of course there may be many plans relevant to a particular event; therefore the further levels of reasoning are required. The next level is 'applicability', which performs a logical check on belief data to determine if a plan is applicable based on the agent's internal state. For example: a 'Found Food Event' is raised when a fish collides with a food source. A logical check is made on the health level of the fish to determine if is below a set level. If the test passes then the 'Eat Food Plan' is applicable, and will be used. This is a different behaviour to acting on a 'Find Food Event' raised when a fish becomes hungry. It is more opportunistic, allowing a fish to eat if it mildly hungry and is in the presence of food, but does not waste food by eating when it has no need to.

The system is able to deal with plan failure and recovery, that is, it will stop trying to execute a plan or achieve a goal when that plan or goal is no longer relevant or applicable. For example, the fish may be hungry, and is executing the 'Find Food Plan'; it has identified a target food source, and is moving in that direction. On route to the food source, another food item is placed in the agent's path, and a 'Found Food Event' is triggered. This event then triggers an 'Update Health Plan' as the plan is both relevant and applicable. The health of the fish agent is thus increased above the trigger threshold for 'Find Food Plan'. This invalidates the plan, and it is subsequently dropped.

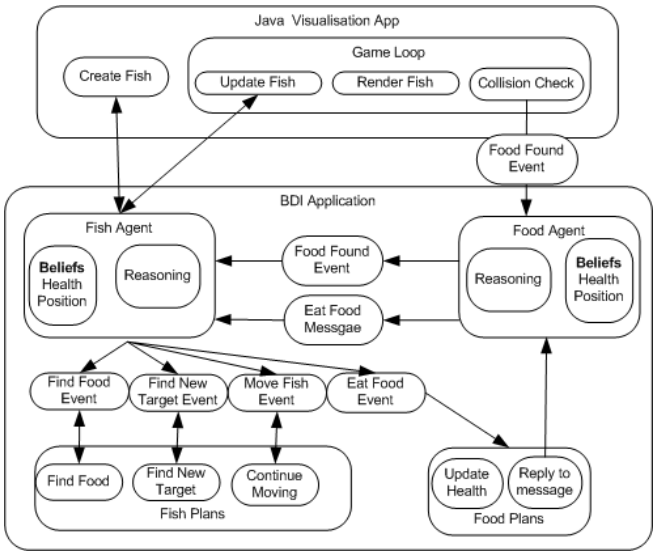

Figure 2 : Fish Tank Architecture

Our fish tank example will continue to run until there is no more food available for the fish, at which point they run out of health and die. It is acknowledged that the agents are quite simplistic at this stage and do not represent more complex behaviour that is exhibited by fish in a fish tank. We could spend more time developing this functionality such as adaptation, memory and learning, as well as the inclusion of systems to deal with a multi-agent environment. However, rather than develop these complex system using the fish tank scenario, we plan to implement our agents in a more sophisticated environment such as a first person shooter computer game where more interesting scenarios can be experimented with. This will introduce a set of complications including linking the AI tool to a graphics engine, and defining the messaging protocol that will be enforced.

VISUALISATION ENGINES

The fish tank example uses a very simple Java based graphics system that paints images on a 2D canvas. A game loop updates the agents, performs a collision check, and repaints the screen. Extra graphical functionality could be developed using Java 3D for example. However, to get more sophisticated engines we have looked at alternatives that stem from the games industry. We have investigated different engine types (Davies et al 2004). Our first attempt to create a visualisation engine was via Microsoft's DirectX API. DirectX provides a set of interfaces for graphics and sound allowing low-level control from high-level interfaces. We created a system for the creation of quasi-accurate 3D scenes that incorporated animated characters. It was possible to create some impressive results in a relatively short space of time and investment. Our implementation loads and orientates 3D models and renders them to screen.

Animated characters are also incorporated running pre-created animation sequences. Users navigate around the scene using mouse and keyboard input to view the action from different locations and angles. However, it becomes increasingly difficult to make significant advances to the graphics engine when requirements become more sophisticated, and our engine started to exhibit limitations quite rapidly. The frame rates for visualisations start to fall below 30fps when more than three characters are present resulting in jerky animations. This is caused by limitations in scene and model optimisation techniques. It is possible to incorporate these features, however, significant allocations of time and resources would be required.

An alternative is the use of commercial game engines such as Quake or Unreal Tournament (Epic Games). These are highly developed systems using modern programming techniques that provide comprehensive sets of tools that allow sophisticated games to be developed. To extend the shelf life of games, many companies provide tools to gaming communities allowing them to customise the original game to make new game types by developing new graphics, physics and game rules. These tools are also available to academia allowing the production of commercial quality visualisations and simulations for a relatively low expenditure of resources (Lewis and Jacobson, (2002)). We have investigated the game engine Unreal Tournament which provides a high performance real time 3D graphics platform. The game comes complete with the mapping tool UnrealEdit that allows game levels to be developed in a graphical environment. Game physics and rules can be customised via the C like language Unreal Script. Each game level consists of game objects, and a network of nodes or waypoints used for navigation. There has been interest in the academic community in the use of Unreal Tournament in research including Sioutis et al (2003) that has arisen partly because of the extension GameBots/JavaBots (Marshall et al., 2004) technology. This is an extension to the Unreal Engine written in Unreal Script that sits between the game engine and an external client that receives and distributes messages via a TCP/IP link. This has the benefit that AI can be written in any language or development environment that allows a network connection.

Using Unreal Tournament, we have managed to develop some relatively complex maps in a short amount of time that performs significantly better than our DirectX engine. However, while sophisticated, using game engines requires us to work within the limits imposed by the game engine developers. The information the game engine is capable of sending and receiving will determine the AI that can be produced. Norling (2004) carried work similar to our research using the Quake engine. They reported a number of limitations where certain behaviours cannot be modelled due to the deficiencies in information sent from the game server. Specifically this related to the behaviour of throwing a grenade at an archway so it would bounce off and hit a chasing bot. As the game engine did not send information identifying when an agent was under an archway, this could not be modelled. It is possible that Unreal Tournament may be able to address some of these issues as the tools provided allow more detailed customisation, i.e. a special node signifying an archway could be incorporated into maps. However, other limitations are sure to exist.

A third solution is to use a commercial or open-source graphics engine rather than a game engine. This would allow us to display complex scenes efficiently, but will not restrict us to conforming to the limits imposed by game developers. Game specific components could then be created via tools, or directly using low-level code. For example, open source physics engines can be added, but the navigation system can be developed internally. There are a plethora of graphics engines available, both open source and commercial. Two engines we have looked at specifically are Torque (Garage Games 2005), and Irrlicht (N Gebhardt. 2005). Irrlicht is a free open-source, cross-platform, 3D graphics engine. It contains most of the features available in commercial game engines including dynamic shadows, character animation, and collision detection. The engine is also platform independent allowing the scenes to be rendered using DirectX, OpenGL, or software interfaces. It also imports graphic file formats from many 3D graphic applications including 3D Studio Max. As the engine is open source, access is possible down to the source level allowing complete control. Other tools are available that plug into Irrlicht, such as the physics engine Tokamak, allowing for sophisticated games to be developed. Similar to Irrlicht, the Torque engine is a fully featured engine that is available for a modest fee. Rather than integrating a suite of different tools, Torque is a complete solution offering tools for graphics, physics and networking. We have experimented with both these tools and accomplished impressive results in a short space of time. It is clear that developing a complete game using these tools is not an easy proposition; however, we can achieve more sophisticated results than attempting to produce a game using low level DirectX code. It is also clear that the final game / simulation will not be as sophisticated as that provided by Unreal Tournament or similar game engine. It will, however, be more accessible for modifications. We have decided to concentrate development using the Irrlicht engine at present as we feel it offers a good balance of customisation and advanced features. However, this need not be the final decision. As the AI is being developed externally it should be possible to link the AI to any game type that allows a network connection.

MESSAGING SYSTEM

All the above solutions offer the potential to create sophisticated 3D game environments that will be a good test bed for our BDI intelligent agents. However, the BDI tool we are using is based in Java and it is therefore not possible to incorporate it directly into these engines which are either closed systems (Unreal Tournament) or C++ based (Irrlicht, Torque). Therefore, a messaging protocol needs to be developed that will allow the exchange of information between the two systems. A good example of this type of architecture is the GameBot system for Unreal Tournament. This system specifies messages into two types; server messages and client messages. Messages from the server provide sensory information detailing what a bot can ‘see’ in the game, as well as information about the bots physical status e.g. health, armour etc. Other information is also sent,

such as if the agent has taken damage, and the direction a shot came from. Messages from the client take the form of commands that instruct the agent to perform tasks such as rotate, walk, and shoot. The GameBots system allows the client to have access to the internal native AI system, and allows instructions such as GETPATH, which returns a list of waypoints between two locations, and RUNTO that will direct the bot to a specified waypoint. Using this system as a template, we intend to develop a messaging system that provides functionality to allow JACK to communicate with our game engine. We will not create a system as comprehensive as that provided by GameBots. Initially we will develop a view system that will return the reachable navigation nodes within a view field in the game and send this information to JACK to update belief set. Periodically, messages will be sent to indicate the agent's position in the environment. Also, messages such as 'Node Reacted' will be sent to identify the completion of a movement task. The agent will send messages to the game such as 'Move to node', and will allow certain information to be queried such as 'Current Position'. Using this simple protocol as a basis, more sophisticated behaviour can be specified in future work.

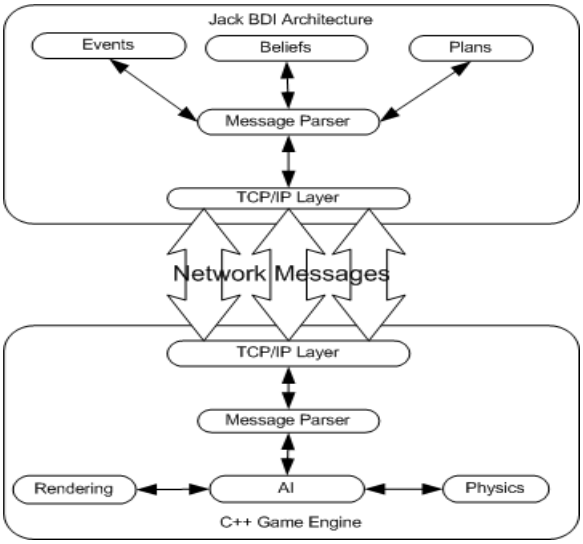

Figure 3 : Messaging Architecture

To send message back and forth various methods of communication protocols exist that facilitate interaction. We have looked at the glue language TCL that have libraries for C++ and Java and allow communication between the platforms. Another approach is to use JACOB, a Java – COM bridge allowing COM processes to be called from a Java component. We have settled on the use of TCP/IP messaging protocol that allows data to be sent back and forth over a network as it is common in the game industry in multi-player games. TCP/IP is also well documented, and simple to develop in both C++ and Java. Currently, we have developed the client and server code for both platforms that allows communication between the two systems. An outline architecture is shown above in Figure 3.

CONCLUSION AND FURTHER WORK

In this paper we have outlined our development of some BDI agents using the JACK agent platform. The agents take the form of fish in a fish tank and are visualised in a simple 2D Java environment. We have identified our intention to implementing our agents in more sophisticated scenarios and have identified alternative ways to visualise the behaviour on various platforms, including high performance 3D graphics engines. It has been identified that using a graphics engine is a good solution as it allows flexibility in game creation, while incorporating sophisticated game programming techniques. This will allow us to concentrate on AI modelling and simulation rather than devoting time for developing a custom graphics engine. We have also identified the need for linking the game engine to a Java based AI engine, and indicated we will use TCP/IP. We also identified the need for creating a messaging protocol specifying which messages will be sent and received over the network. Further work will include the development of the messaging system, and the construction of more sophisticated environments / games in which the agents will be situated. We will also begin to incorporate more complex concepts such as teamwork, emotion and adaptation.

REFERENCES

Agent Oriented Software Pty. Ltd. JACK Intelligent Agents. http://www.agent-software.com

Bratman, M. (1987) "Intentions, plans and practical reasoning." Harvard University Press, Cambridge, Massachusets.

Braubach, L., Pokahr, A., 'JADEX' http://vsis-www.informatik.uni-hamburg.de/projects/jadex/ Accessed 10.03.2005

Davies, N.P., Mehdi, Q.H., and Gough, N.E (2005) 'Creating and Visualising an Intelligent NPC using Game Engines and AI Tools', *Proc 19th European Conference on Modelling and Simulation, ECMS 2005,* Riga, Latvia, 721-726, ISBN 1-84233-112-4

Epic Games Inc. Unreal Tournament. http://www.unreal.com

Gebhardt, N (2005) Irrlicht Engine and Irrlicht Engine http://irrlicht.sourceforge.net/

Norling, E. (2004) "Folk psychology for human modelling: extending the BDI paradigm". In : Int. Conf. on Autonomous Agents and Multi Agent Systems (AAMAS), New York, 2004.

Lewis, M., Jacobson, J., (2002), 'Game Engines in Scientific Research', Communications of the ACM, Volume 45, Issue 1. New York, USA

Rao, A., Georgeff, M., (1995) "BDI agents: from theory to practice" Tech. Rep. 56, Autralian Artificial Intelligence Institute, Melbourne, Australia

Sioutis, C, Ichalkaranje, N & Jain, LC 2003, 'A framework for interfacing BDI Agents to a real-time simulated environment', Design and Application of Hybrid Intelligent Systems, A Abraham, M Koppen & K Franke (eds), proceedings of the 3rd International Conference of Hybrid Intelligent Systems (HIS'03), Melbourne, Australia, December 2003, IOS-Press, Amsterdam, The Netherlands, pp. 743-748.

TiLab, Java Agent Development Environment (JADE) http://jade.tilab.com/ accessed 10/3/2005