QASE: AN INTEGRATED API FOR IMITATION AND GENERAL AI RESEARCH IN COMMERCIAL COMPUTER GAMES

Bernard Gorman
Dublin City University
Glasnevin, Dublin 9
Rep. of Ireland
+353 1 4902714
bernard.gorman@computing.dcu.ie

Martin Fredriksson
Blekinge Institute of Technology
Ronneby
Sweden
+46 457 385825
martin.fredriksson@bth.se

Mark Humphrys
Dublin City University
Glasnevin, Dublin 9
Rep. of Ireland
+353 1 700 8059
mark.humphrys@computing.dcu.ie

KEYWORDS

Imitation, machine learning, artificial intelligence, API, game bots, intelligent agents, education.

ABSTRACT

Computer games have belatedly come to the fore as a serious platform for AI research. Through our own experiments in the fields of *imitation learning* and intelligent agents, it became clear that the lack of a unified, powerful yet intuitive API was a serious impediment to the adoption of commercial games in both research and education. Parallel to our own specialised work, we therefore decided to develop a general-purpose library for the creation of game agents, in the hope that the availability of such software would help stimulate further interest in the field. Though geared towards machine-learning, the API would be flexible enough to facilitate multiple forms of artificial intelligence, making it suitable for application in research and in undergraduate courses centring upon traditional AI and agent-based systems.

In this paper, we present the result of our efforts; the Quake 2 Agent Simulation Environment (QASE) API. We first describe the theme of our work, the reasons for choosing Quake 2 as our testbed, and the necessity for an API of this nature. We then outline its most important features, before presenting an experiment from our own research to demonstrate QASE's practical capabilities.

INTRODUCTION

In recent years, commercial computer games have gained increasing recognition as an ideal platform for research in various fields of artificial intelligence (Larid & van Lent, 2000; Naraeyek 2004). The vast majority, however, still utilize AI techniques that were developed several decades ago, and which often produce mechanical, repetitive and unsatisfying game agents. Given that games provide a convenient means of recording the complex, fluent behaviors of human players, some researchers (Sklar et al 1999; Bauckhage et al 2003; Thurau et al 2004) have speculated that approaches based on the analysis and *imitation* of human demonstrations may produce more challenging and believable artificial agents than can be realised using traditional techniques; indeed, imitation learning is already employed quite extensively in the robotics community (Atkeson & Schaal 1997, Schaal 1999, Jenkins & Mataric 2000). Building upon this premise, the primary focus of our work lies in investigating imitation learning in games which involve cognitive agents.

In the initial stages of our research, however, it became clear that the available testbeds and resources were often scattered, frequently incomplete, and consistently ad hoc. Existing APIs were unintuitive, unreliable and lacking in functionality. Network protocol and file format specifications were generally unofficial, more often than not the result of reverse-engineering by adventurous fans (Girlich 2000). Documentation was sketchy, with even the most rudimentary information spread across several disjoint sources. Above all, it was evident that the absence of a unified, low-level yet easy-to-use development platform and experimental testbed was a major impediment to the adoption of commercial games in both academic research and education.

As a result, we decided to adopt a two-track approach. We would develop approaches to imitation learning in games, while simultaneously building a comprehensive programming interface designed to provide all the functionality necessary for others to engage in this work. This interface should be powerful enough to facilitate high-end research, while at the same time being suitable for use in undergraduate courses geared towards classic AI and agent-based systems.

Choosing a Testbed - Quake 2

Our first task was to decide which game to use as a testbed. We opted to investigate the *first-person shooter* genre, in which players control a single character exploring a three-dimensional environment littered with weapons, bonus items, traps and pitfalls, with the objective of defeating as many opponents as possible within a predetermined time limit. This particular genre was chosen in preference to others due to the fact that it provides a comparatively *direct* mapping of human decisions onto agent actions; this is in contrast to many other game types, where the agent's behaviours are determined in large part by factors other than the player's decision-making process. In sports simulations, for instance, only a single character is usually under the control of the human player - the interactions of his teammates are managed from one timestep to the next by the computer. While other genres do offer many interesting challenges for AI research, as outlined by both (Laird 2001) and (Fairclough et al 2001), the attraction of first-person shooters - to researchers and gamers alike - lies in the minimal degree of abstraction they impose between the human player and his/her virtual avatar. The same qualities make them ideal for use in undergraduate courses; the student creates the AI for a single agent, which can then be deployed in competition against those written by others.

With this in mind, we chose ID Software's **Quake 2** as our test environment - it was prominent in the literature, existing resources were more substantial than for other games, and thanks to Laird it had become the de facto standard for research of this nature. **Figure 1** shows a typical Quake 2 environment, with various features labelled.

THE QASE API

The Quake 2 Agent Simulation Environment was created to meet the requirements identified earlier; namely, it is a fully-featured, integrated API, designed to be as intuitive, modular and transparent as possible. It is Java-based, ensuring an easily extensible object-oriented architecture and allowing it to be deployed on many different hardware platforms and operating systems. It amalgamates and improves upon the functionalities of several existing applications, removing the need to rely on ad-hoc software combinations or to comb through a multitude of different documentations; QASE consolidates all relevant information into a single source. It is geared towards machine and imitation learning, but is equally appropriate for use with more traditional forms of agent-based AI. Put simply, QASE is intended to provide all the functionality the researcher or student will require in their experiments with cognitive agents.

In the following sections we will outline the major components of the QASE architecture, highlighting its potential for application in research and education.

Network Layer

Quake 2's multi-player mode is a simple client-server model. One player starts a server and other combatants connect to it, entering whatever environment (known as a *map*) the instigating player has selected. Every hundred milliseconds, the server transmits an update frame to all connected clients, containing information about the game world and the status of each entity; each client merges the update into its existing gamestate record, and then responds by sending its desired movement, aiming and action back to the server. Thus, in order to realize artificial agents (also known as *bots*), a means of handling the game's network traffic is required.

QASE accomplishes this via its *Proxy* class, which encapsulates an implementation of the Quake 2 client-side network protocol. It is responsible for establishing game sessions with the server, receiving inbound data and converting it into a human-readable format, and transmitting the agent's subsequent actions back to the server, as shown in **Figure 2** below. All this is transparent to the agent itself; at each interval, the bot is simply notified that an update has occurred, and receives a *World* object containing a hierarchy of component objects representing the current gamestate.

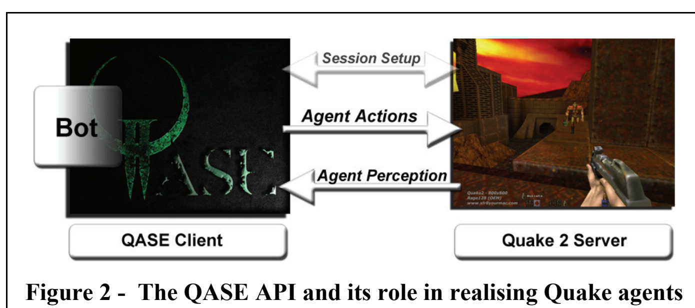

Figure 2 - The QASE API and its role in realising Quake agents

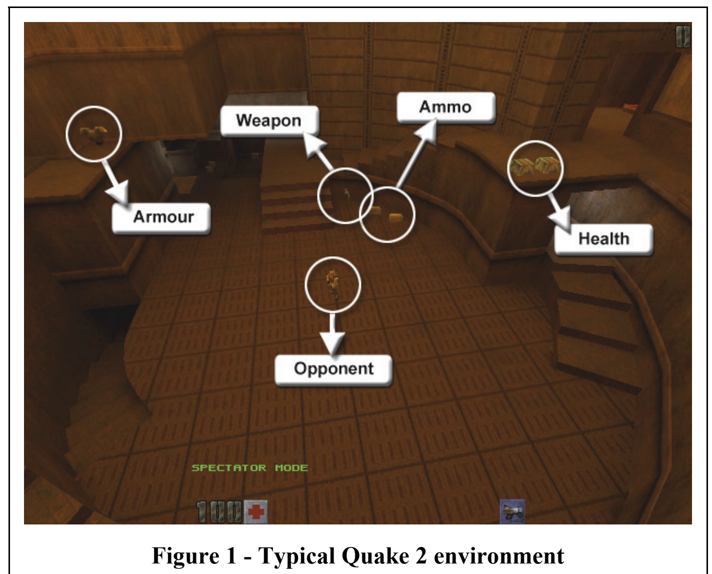

Figure 1 - Typical Quake 2 environment

An important point to note is that, because the network layer is separated from the higher-level classes in the QASE architecture, it is highly *portable*. Adapting the QASE API to games with similar network protocols, such as Quake 3 and its derivatives, therefore becomes a relatively straightforward exercise; by extending the existing classes and rewriting the data-handling routines, they could conceivably be adapted to any UDP-based network game. Thus, QASE's network structures can be seen as providing a *template* for the development of artificial game clients in general.

Gamestate Augmentation

Rather than simply providing a bare-bones implementation of the client-side protocol, QASE also performs several behind-the-scenes operations upon receipt of each update, designed to present an *augmented* view of the gamestate to the agent. In other words, QASE transparently analyses the information it receives, makes deductions based on what it finds, and exposes the results to the agent. As such, it may be seen as representing a *virtual extension* of the standard Quake 2 network protocol.

For instance, the standard protocol has no explicit *item pickup* notification; when the agent collects an object, the server takes note of it but does not send a confirmation message to the client, since under normal circumstances the human player will be able to identify the item visually. QASE compensates for this by detecting the sound of an item pickup, examining which entities have just become inactive, finding the closest such entity to the player, and thereby deducing the entity number, type and inventory index of the newly-acquired item. Building on this, QASE records a full list of which items the player has collected and when they are due to *respawn* (reappear), automatically flagging the agent whenever such an event occurs.

Similarly, recordings of Quake 2 matches (see below) do not encode the full inventory of the player at each timestep - that is, the list of how many of which items the player is currently carrying. For research models which require knowledge of the inventory, such as that outlined in the

QASE and Imitation Learning section below, this is a major drawback. QASE circumvents the problem by monitoring item pickups and weapon discharges, 'manually' building up an inventory representation from each frame to the next. This can also be used to track the agent's inventory in online game sessions, removing the need to explicitly request a full inventory listing from the server on each update.

Bot Hierarchy

In order to facilitate the rapid creation of different types of game agents, QASE implements a structured *hierarchy* of bot classes, allowing users to develop agents from a number of levels of abstraction. These range from a simple interface class, to full-fledged bots incorporating an exhaustive range of user-accessible functions. The bot hierarchy comprises three major levels; these are summarised below.

Bot

A template which specifies the interface to which all bots must conform, but does not provide any functionality; the programmer is entirely responsible for the implementation of the agent, and may do so in any way (s)he chooses.

BasicBot

An abstract bot which provides most of the functionality required by Quake 2 agents, such as the ability to determine whether the bot has died, to respawn (re-enter the game) after the agent has been defeated, to create an agent given minimal profile information, to set the agent's movement direction, speed and aim and send these to the server, to obtain sensory information about the virtual world, and to record itself to a demo file. All that is required of the programmer is to extend the class, write the AI routine in the predefined `runAI` method, and to supply a means of handling the server traffic according to whatever interaction paradigm he wishes to use. The third level of the bot hierarchy provides ready-to-use implementations of two such paradigms.

ObserverBot and PollingBot

The highest level of the Bot hierarchy consists of two classes, `ObserverBot` and `PollingBot`, which represent fully-realised agents. Each of these provides a means of detecting changes to the gamestate (implemented as indicated by their names), as well as a single point of insertion - the programmer needs only to supply the AI routine in the `runAI` method defined by the Bot interface. Each has its own advantages; the `ObserverBot` allows several different objects to be attached to a single Proxy, whereas the multithreaded `PollingBot` offers slightly more efficient performance.

Beyond this, several convenience classes are available, which provide extended bot implementations tailored to specific purposes. The `NoClipBots` allow the user to 'noclip' the agent (i.e. move it through otherwise solid walls) to any arbitrary point in the environment before starting the simulation; the `MatLabBot` branches will be explained later. The full hierarchy is shown in **Figure 3** below.

The DM2 Parser and Recorder

Quake 2's inbuilt client, used by human players to connect to the game server, facilitates the recording of matches from the perspective of each individual player. These *demo* or *DM2* files contain an edited copy of the network packet stream received by the client during the game session, capturing the player's actions and the state of all entities at each discrete time step. For the purposes of imitation learning, then, a means of parsing these files and extracting the gameplay samples is needed. QASE's *DM2Parser* fulfils this requirement.

The *DM2Parser* treats the demo file as a virtual server, "connecting" to it and reading blocks of data in exactly the same manner as it receives network packets during an online game session. A copy of the gamestate is returned for each recorded frame, and the programmer may query it to retrieve whatever information (s)he requires.

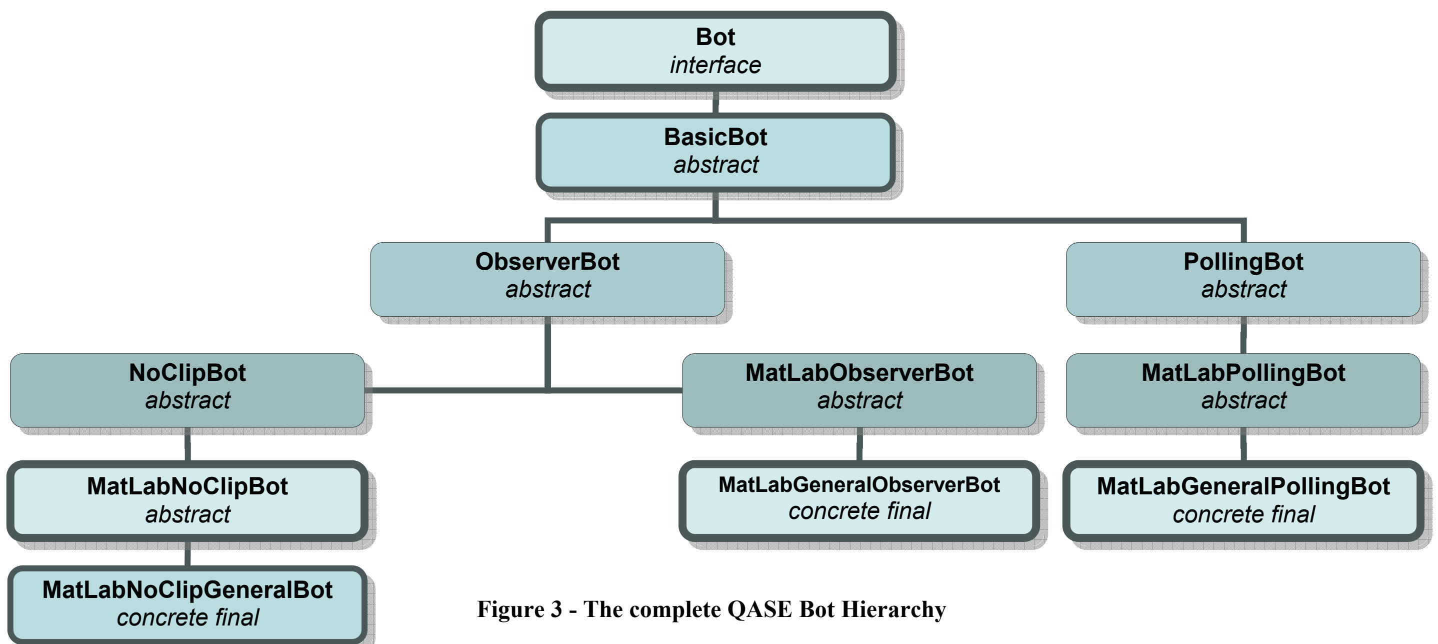

Figure 3 - The complete QASE Bot Hierarchy

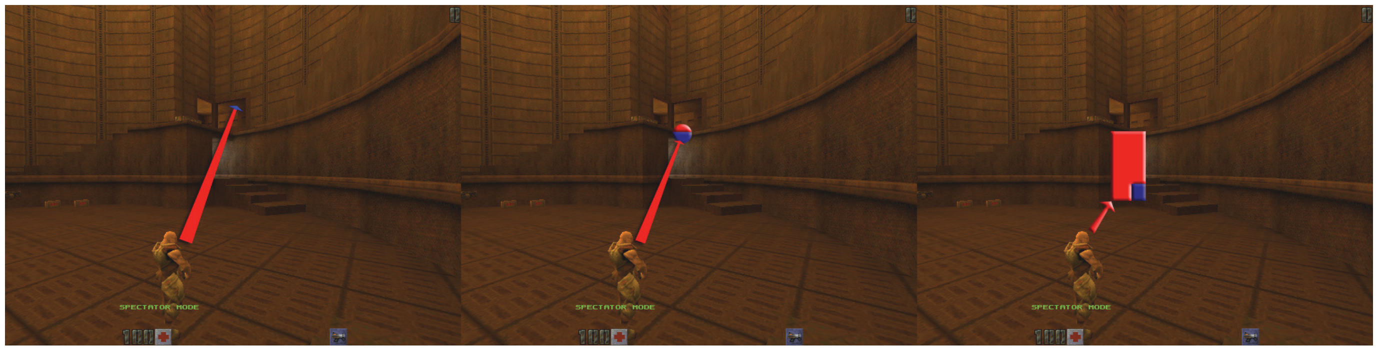

Figure 4 - BSP traces with line, sphere and box. Collision occurs at different points.

For examples of the type of data that can be obtained and analysed, see the sections *MatLab Integration* and *QASE and Imitation Learning* below.

Furthermore, QASE incorporates a *DM2Recorder*, allowing the agent to automatically record a demo of itself during play; this actually improves upon Quake 2's standard recording facilities, by allowing demos spanning multiple maps to be recorded in playable format. The incoming network stream is sampled, edited as necessary, and saved to file when the agent disconnects from the server or as an intermediate step whenever the map is changed.

Environment Sensing

The network packets received by game clients from the Quake 2 server do not encode any information about the actual environment in which the agent finds itself, beyond its current state and those of the various game entities present. This information is contained in *Binary Space Partition* (*BSP*) files stored locally on each client machine; thus, in order to provide the bot with more detailed sensory information (such as determining its proximity to an obstacle, or whether an enemy is visible), a means of locating, parsing and querying these map files is required. QASE's *BSPParser* and *PAKParser* fulfil this need.

The BSP file corresponding to the active map in the current game session may be stored in the default game directory, a custom game directory, or in any of Quake 2's *PAK* archives; its filename may or may not match the name of the map, which is the only information possessed by the client. If the user sets an environment variable pointing to the location of the base Quake 2 folder, QASE can automatically find the relevant BSP by searching each location in order of likelihood. This is done transparently from the agent's perspective; as soon as any environment-sensing method is invoked, the map is silently located, loaded and queried.

Once loaded, the *BSPParser* can be used to sweep a **line**, **box** or **sphere** in any arbitrary direction through the game world, starting from the agent's current location; the distance and/or position at which the first collision with the environment's geometry occurs is returned. This allows the agent to "perceive" the world around it on a pseudo-visual level - line traces can be used to determine whether entities are visible from the agent's perspective, sphere traces can be used to check whether projectiles will reach a certain point if fired, and box traces can be used to determine whether the agent's in-game model will fit through an opening. **Figure 4** above shows the operation of each different trace type.

Inbuilt Cognitive & Other Facilities

For education purposes, QASE incorporates implementations of both a *neural network* and a *genetic algorithm generator*. These are designed to be used in tandem - that is, the genetic algorithms gradually cause the neural network's weights to evolve towards a given fitness function. A *KMeans* calculator class is also included; aside from serving as an illustration of clustering techniques, it is also used in QASE's *waypoint map generator* (see below). These features are included primarily to allow students to experiment with some AI constructs commonly found in undergraduate curricula - for more demanding research applications, QASE allows MatLab to be used as a back-end.

One of QASE's most useful features, particularly from an educational point of view, is the aforementioned *waypoint map generator*. Drawing on concepts developed in the course of our work in imitation learning (see *QASE and Imitation Learning*), this requires the user to supply a prerecorded DM2 file; it will then automatically find the set of all positions occupied by the player during the game session, cluster them to produce a smaller number of indicative *waypoints*, and draw *edges* between these waypoints based on the observed movement of the demonstrator. The items collected by the player are also recorded, and Floyd's algorithm (Floyd, 1962) is applied to find the matrix of distances between each pair of points. The map returned to the user at the end of the process can thus be queried to find the shortest path from the agent's current position to any needed item, to the nearest opponent, or to any random point in the level. Rather than manually building a waypoint map from scratch, then, all the student needs to do in order to create a full navigation system for their agent is to record themselves moving around the environment as necessary, collect whatever items their bot will require, and present the resulting demo file to QASE.

MatLab Integration

For the purposes of our work in imitation learning, we need a way to not only obtain, but also statistically analyse the observed in-game actions of human players. Rather than hand-coding the required structures from scratch, we opted instead to integrate the API with the Mathworks™ MatLab®

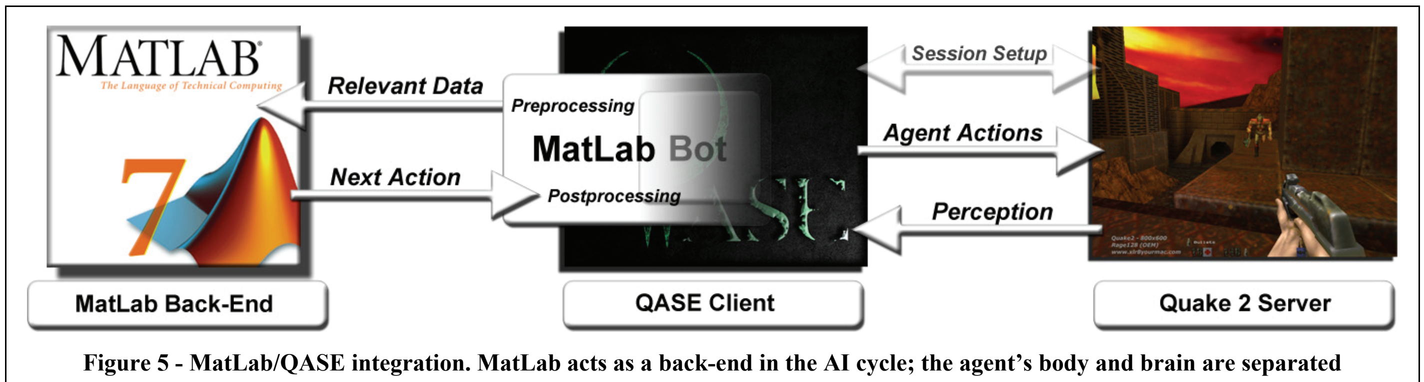

Figure 5 - MatLab/QASE integration. MatLab acts as a back-end in the AI cycle; the agent's body and brain are separated

programming environment. Given that it provides a rich set of built-in toolboxes for neural computation, clustering and other classification techniques and is already widely used in research, MatLab seemed an ideal choice to act as an optional back-end for QASE agents.

Bots can be instantiated and controlled via MatLab in one of two ways. For simple AI routines, one of the standalone `MatLabGeneralBots` shown in **Figure 3** is sufficient. A MatLab function is written which creates an instance of the agent, connects it to the server, and accesses the gamestate at each update, all entirely within the MatLab environment. The advantage of this approach is that it is intuitive and very straightforward; a template of the MatLab script is provided with the QASE API. In cases where a large amount of gamestate and data processing must be carried out on each frame, however, handling it exclusively through MatLab can prove somewhat inefficient.

For this reason, we developed an alternative paradigm designed to offer greater efficiency. As outlined in the *Bot Hierarchy* section above, QASE agents are usually created by extending either the `ObserverBot` or `PollingBot` classes, and overloading the `runAI` method in order to add the required behaviour. In other words, the agent's AI routines are *atomic*, and encapsulated entirely within the derived class. Thus, in order to facilitate MatLab, a new branch of agents - the `MatLabBots` - was created; each of these possesses a three-step AI routine as follows:

1. On each server update, QASE first *pre-processes* the data required for the task at hand; it then flags MatLab to take over control of the AI cycle.
2. The MatLab function obtains the agent's input data, processes it using its own internal structures, passes the results back to the agent, and signals that the agent should reassume control.
3. This done, the bot applies MatLab's output in a *postprocessing* step.

This framework is already built into QASE's `MatLabBots`; the programmer need only extend `MatLabObserver / Polling / NoClipBot` to define the handling of data in the preprocessing and postprocessing steps, and change the accompanying MatLab script as necessary. By separating the agent's ***body*** (QASE) from its ***brain*** (MatLab) in this manner, we ensure that both are modular and reusable, and that cross-environment communications are minimised. The preprocessing step filters the gamestate, presenting only the minimal required information to MatLab; QASE thus enables both MatLab and Java to process as much data as possible in their respective native environments. This has proven extremely effective, both in terms of computational efficiency and ease of development.

Aside from creating game agents, MatLab can also use the various supporting functions of the QASE API. From our perspective, one of the most important of these is the ability to read and process demonstrations of gameplay using the *DM2Parser*. **Figure 8** shows an example of this; see the section *QASE and Imitation Learning* for details.

Of course, the fact that we integrated QASE with MatLab specifically to facilitate our work in imitation learning does not diminish its potential for use in other areas; as stated earlier, QASE is designed for broad AI research.

QASE AND IMITATION LEARNING

In this section, we outline an experiment conducted in the course of our work. While it by no means demonstrates the full extent of QASE's faculties, this example does provide a good indication of its potential in the field of research.

One of the first questions which arises when considering the problem of imitation learning is, quite simply, "what behaviours does the demonstration encode?" To this end, (Thurau et al 2004a) propose a model of in-game behaviour based closely on Hollnagel's COCOM (Hollnagel 1993), as shown in **Figure 6** below.

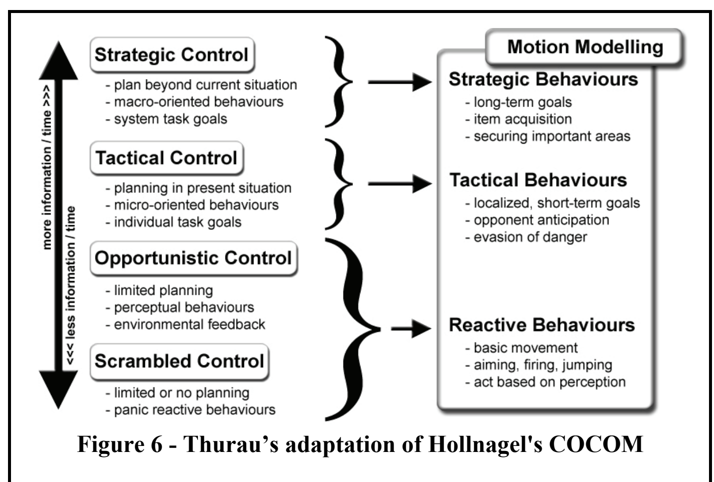

Figure 6 - Thurau's adaptation of Hollnagel's COCOM

Strategic behaviours refer to actions the player takes with long-term goals in mind; these include maximising the number of weapons or items he possesses, controlling certain areas of the map, and so forth. *Tactical* behaviours are mostly concerned with localised tasks such as evading or engaging opponents. *Reactive* behaviours involve little or no planning; the player simply reacts to stimuli in his immediate surroundings. *Motion modelling* refers to the imitation of the player's movement; in theory, this should produce *humanlike* motion along the bot's path, and should also prevent the agent from performing actions which are impossible for the human player's mouse-and-keyboard interface (instantaneous 180˚ turning, perfect aim, etc).

Goal-Oriented Strategic Behaviour

The following is drawn largely from our paper *"Towards Integrated Imitation of Strategic Planning and Motion Modelling in Interactive Computer Games"* (Gorman & Humphrys 2005).

In order to learn long-term strategic behaviours from human demonstration, we developed a model designed to emulate the notion of *program level imitation* discussed in (Byrne and Russon 1998); in other words, to identify the demonstrator's *intent*, rather than simply reproducing his precise *actions*. (Thurau et al, 2004a) present an approach to such behaviours based on artificial potential fields; here we consider the application of reinforcement learning and fuzzy clustering techniques.

Topology Learning

As mentioned earlier, in the context of Quake, strategic planning is mostly concerned with the efficient collection and monopolisation of *items* and the control of certain important areas of the map. With this in mind, we first read the set of all player locations $\vec{l} = \{x, y, z\}$ from the DM2 recording into MatLab via QASE's *DM2Parser*, and the points are clustered to produce a reduced set of positions, called *nodes*. We initially employed the Neural Gas algorithm in this step, since it has been demonstrated to perform well in topology-learning tasks (Martinez et al 1993); however, we later developed a custom modification of Elkan's *fast k-means* (Elkan 2003) designed to treat the positions at which items were collected as immovable *"anchor"* centroids, thereby deriving a goal-oriented clustering of the dataset. By examining the sequence of player positions, we also construct an *n x n* matrix of edges *E*, where *n* is the number of clusters, and $E_{ij} = 1$ if the player was observed to move from node *i* to node *j* and 0 otherwise.

Deriving Movement Paths

Because the environment described above may be seen as a Markov Decision Process, with the nodes corresponding to states and the edges to transitions, we chose to investigate approaches to goal-oriented movement based on concepts from *reinforcement learning*, in particular the *value iteration* algorithm.

To do so, we first read the player's inventory from the demo at each timestep, again using QASE's *DM2Parser* and the inventory-tracking system described earlier. In our experiments, we construct an inventory state vector of 18 elements, specifying the player's *health* and *armour* values together with the *weapons* he has collected and the amount of *ammo* he has for each. The set of unique state vectors is then obtained; these state prototypes represent the varying situations faced by the player during the game session.

We can now construct a set of *paths* which the player followed while in each inventory state. These paths consist of a series of transitions between clusters:

$$t_i = [c_{i,1}, c_{i,2}, ..., c_{i,k}]$$

where t_i is a transition sequence (path), and $c_{i,j}$ is a single node along that sequence. Each path begins at the point where the player enters a given state, and ends where he exits that state - in other words, when an item is collected that causes the player's inventory to shift towards a different prototype. See **Figure 8** for an illustration of one such path.

Assigning Rewards

Having obtained the different paths pursued by the player in each inventory state, we turn to reinforcement learning to reproduce his behaviour. In this scenario, the MDP's actions are considered to be the *choice to move to a given node from the current position*. Thus, the transition probabilities are

$$P(s' = j \mid s = i, a = j) = E_{ij}$$

To guide the agent along the same routes taken by the player, we assign an increasing reward to consecutive nodes in each path taken in each prototype, such that

$$R(p_i, c_{i,j}) = j$$

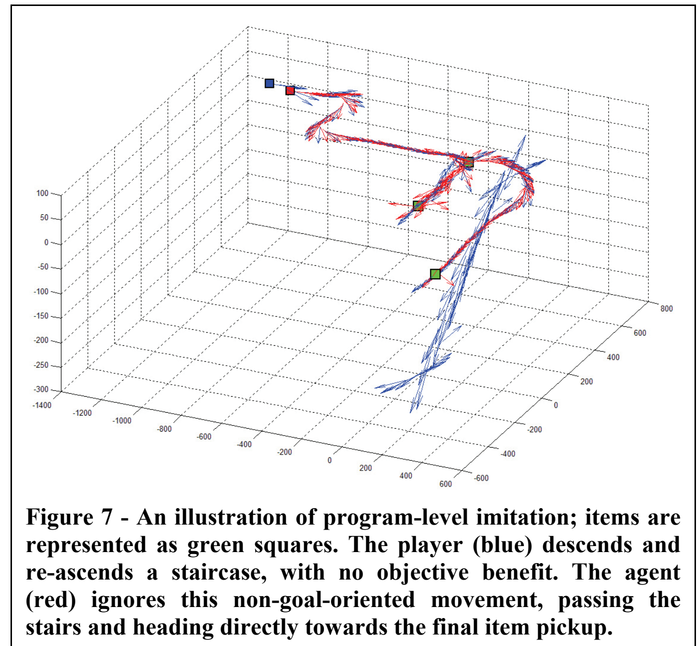

Figure 7 - An illustration of program-level imitation; items are represented as green squares. The player (blue) descends and re-ascends a staircase, with no objective benefit. The agent (red) ignores this non-goal-oriented movement, passing the stairs and heading directly towards the final item pickup.

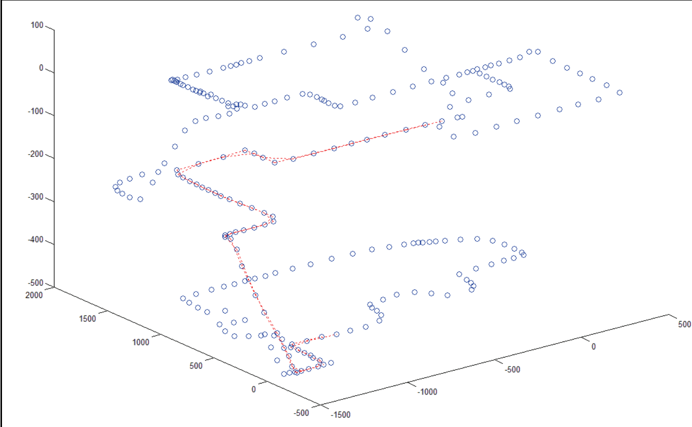

Figure 8 - An example of a path followed by the player while in a particular inventory state. The path originates in the lower part of the level, and ends at the point where the player picked up an item that caused his inventory to shift towards another prototype.

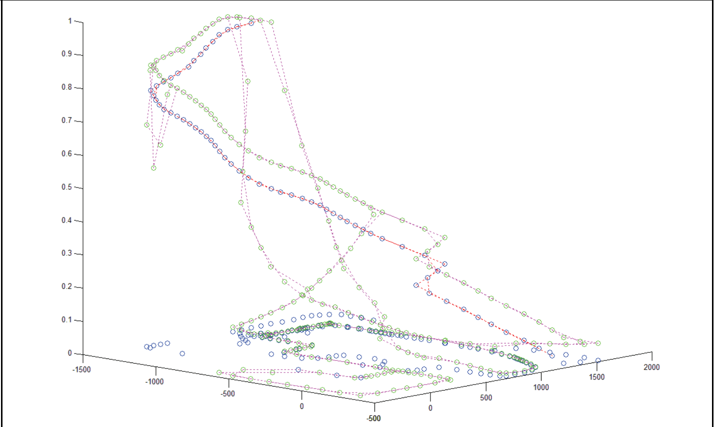

Figure 9 - The ascending rewards assigned to this path (blue/red), and the results of the value iteration algorithm (green & magenta). The y-axis deontes the values associated with each waypoint in the topological map.

where p_i is a prototype, and $c_{i,j}$ is the jth cluster in the associated movement sequence. Each successive node along the path's length receives a reward greater than the last, until the final cluster (at which an inventory state change occurred) is assigned the highest reward. If a path loops back or crosses over itself en route to the goal, then the higher values will overwrite the previous rewards, ensuring that the agent will be guided towards the terminal node while ignoring any non-goal-oriented diversions. Thus, as mentioned above, the agent will emulate the player's program-level behaviour, instead of simply duplicating his exact actions. See **Figure 7** above for an example.

Learning Utility Values

With the transition probabilities and rewards in place, we can now run the value iteration algorithm in order to compute the utility values for each node in the topological map under each inventory state prototype. The value iteration algorithm iteratively propagates rewards outwards from terminal nodes to all others, discounting them by distance from the reward signal; once complete, these utility values will represent the "usefulness" of being at that node while moving to the goal.

In our case, it is important that *every* node in the map should possess a utility value under *every* state prototype by the end of the learning process, thereby ensuring that the agent will always receive strong guidance towards its goal. We adopt the *game value iteration* approach outlined in (Hartley et al 2004) - the algorithm is applied until all nodes have been affected by a reward at least once. **Figure 9** above shows the results of the value iteration algorithm on a typical path.

Multiple Weighted Objectives

Faced with a situation where several different items are of strategic benefit, a human player will intuitively *weigh* their respective importance before deciding on his next move. To model this, we adopt a *fuzzy* clustering approach. On each update, the agent's current inventory is expressed as a membership distribution across all prototype inventory states. This is computed as:

$$m_p(s) = \frac{d(\vec{s},\vec{p})^{-1}}{\sum_{i=1}^{P} d(\vec{s},\vec{i})^{-1}}$$

where s is the current inventory state, p is a prototype inventory state, P is the number of prototypes, d^{-1} is an inverse-distance or proximity function, and $m_p(s)$ is the degree to which state vector s is a member of prototype p, relative to all other prototypes. The utility configurations associated with each prototype are then weighted according to the membership distribution, and the adjusted configurations *superimposed*; we also apply an *online discount* to prevent the possibility of backtracking. The formula used to compute the final utilities is thus:

$$U(c) = \gamma^{e(c)} \sum_{p=1}^{P} V_p(c) m_p(s)$$

$$c_{t+1} = \max_{y} U(y),\ y \in \{x \mid E_{c_t x} = 1\}$$

where $U(c)$ is the final utility of node c, γ is the online discount, $e(c)$ is the number of times the player has entered cluster c since the last state transition, $V_p(c)$ is the original value of node c in state prototype p, and E is the edge matrix.

Object Transience

Another important element of planning behaviour is the human's understanding of *object transience*. A human player intuitively tracks which items he has collected from which areas of the map, can easily estimate when they are scheduled to reappear, and adjusts his strategy accordingly. In order to capture this, we introduce an *activation* variable in the computation of the membership values; inactive items are nullified, and the membership values are redistributed among those items which are still active.

$$m_p(s) = \frac{a(o_p) d(\vec{s},\vec{p})^{-1}}{\sum_{i=1}^{P} a(o_i) d(\vec{s},\vec{i})^{-1}}$$

where a, the *activation* of an item, is 1 if the object o at the terminal node of the path associated with prototype state p is present, and 0 otherwise.

Figure 10 - The agent returns to a previously-visited point before some ammo items have respawned (1.1), and since they are inactive it initially passes by (1.2); however, their sudden re-emergence (1.2) causes the utilities to reactivate, and the agent is drawn to collect them (1.3) before continuing (1.4). Later, the agent returns once again (2.1). The items are now active, but since the agent has already collected several shotgun pickups, the relevant membership values are insignificant; as a result, the agent ignores the pickups (2.2, 2.3), and continues on towards more attractive objectives (2.4)

Deploying the Agent

With the DM2 data extracted and the required values computed, we can now deploy the agent. We extend any of the `MatLabBots`, overloading `preMatLab` to extract the player's current position and inventory and pass these to MatLab. We then rewrite the MatLab template to instantiate the agent and connect it to the server. On each update, MatLab determines the closest matching state prototype and node, extracts the relevant utility configuration, finds the set of nodes connected to the current node by examining the edge matrix, and selects the successor with the highest utility value; the position of this node is passed back to QASE. The agent's `postMatLab` method is also overloaded, to determine the direction between its current position and the next node, and to set the agent's movement accordingly. As the agent traverses its environment, item pickups and in-game events will cause its inventory to change, resulting in a corresponding change in the utility values and attracting the agent towards its next objective. **Figure 10** shows the QASE agent in action.

CONCLUSION

In this paper, we identified the lack of a fully-featured, consolidated API as a major impediment to the adoption of commercial games in AI education and research. We then presented our QASE API, which has been developed to meet these requirements. Several of its more important features were described, and their usefulness highlighted. A practical demonstration of QASE as it has been used in our own research closed this contribution.

FUTURE WORK

Although we regard it as being very feature-rich and entirely stable at this point, QASE will continue to develop as we progress in our research. The two tracks of our work - that of investigating approaches to imitation learning and of building an accompanying API - have thus far informed each other; as mentioned earlier, QASE's waypoint generator is derived from the approach outlined in the section *QASE and Imitation Learning*. In this way, further developments in our research will guide future development of the API.

QASE has already attracted some attention in academia; researchers at Kyushu University in Japan expressed interest in adopting it for use in their work, and more recently a PhD student in California has contacted us with the same intent. As more individuals and institutions discover QASE, the resulting feedback will aid us in continually improving the API. We hope that this paper will help to stimulate further interest in QASE, in imitation learning, and in the potential of games in AI research and education in general.

To download the API and accompanying documentation, please visit the QASE homepage: http://qase.vze.com

REFERENCES

- Bauckhage C, Thurau C. & Sagerer G. (2003): Learning Humanlike Opponent Behaviour for Interactive Computer Games, Pattern Recognition, Vol 2781
- Byrne, R.W. and Russon, A.E. "Learning by Imitation: A Hierarchical Approach", Behavioral and Brain Sciences (1998) 21, 667-721
- Elkan, C. "Using the Triangle Inequality to Accelerate k-Means", Proc. 20th International Conference on Machine Learning 2003
- Fairclough, C., Fagan, M., MacNamee, B and Cunningham, P. Research Directions for AI in Computer Games. Technical report, 2001.
- Floyd, R.W., Algorithm 97, Shortest path, Comm. ACM. 5(6), 1962, 345
- Girlich, U., Unofficial Quake 2 DM2 Format Description, 2000
- Gorman, B & Humphrys, M: "Towards Integrated Imitation of Strategic Planning and Motion Modelling in Interactive Computer Games", Proc. 3rd Intl. Conf. in Computer Game Design & Technology, GDTW05, pages 92-99
- Hartley, T, Mehdi, Q and Gough, N. "Applying Markov Decision Processes to 2D Real-Time Games", Proc. CGAIDE 2004: p55-59
- Hollnagel, E. (1993) Human reliability analysis: Context and control. L:AP
- Jenkins, OC and Mataric, MJ "Deriving Action and Behavior Primitives from Human Motion Data". Proc. IEEE/RSJ IROS-2002, pages 2551-2556
- Laird, J. E. and v. Lent, M. (2000). Interactive Computer Games: Human-Level AI's Killer Application. AAAI, pages 1171-1178.
- Laird, J.E. Using a Computer Game to develop advanced AI. IEEE Computer, pages 70 -75, July 2001.
- Martinez, T. and Schulten, K. (1991). A neural gas network learns topologies. In Artificial Neural Networks. Elseviers Science Publishers
- Naraeyek, A. Computer Games - Boon or Bane for AI Research. Künstliche Intelligenz, pages 43-44, February 2004
- Schaal, S. Is imitation learning the route to humanoid robots? Trends in Cognitive Sciences, 3(6):233-242, 1999
- Sklar, E., AD Blair, P Funes & J. Pollack, 1999: Training Intelligent Agents Using Human Internet Data, 1st Asia-Pacific IAT
- Thurau, C., C. Bauckhage & G. Sagerer 2004a: Learning Humanlike Movement Behaviour for Computer Games, in Proc. 8th Intl. SAB Conf.
- Thurau, C., C. Bauckhage & G. Sagerer 2004b: Synthesising Movement for Computer Games, in Pattern Recognition, Vol. 3175 of LCNS Springer