THE QASE API: AN INTEGRATED PLATFORM FOR AI RESEARCH AND EDUCATION THROUGH FIRST-PERSON 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, Quake.

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 comparing QASE against other game-based artificial intelligence APIs. A presentation of some experiments from our own research demonstrating QASE's practical capabilities closes this contribution.

INTRODUCTION

In recent years, commercial computer games have gained increasing recognition as an ideal platform for research in various fields of artificial intelligence (Laird & 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 generally ad hoc. Existing APIs were unintuitive, unreliable and lacking in functionality. Network protocol and file format specifications were usually 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 real-time strategy games, for instance, the player typically controls a large number of units, directing them in various scheduling and resource management tasks; although the player is responsible for, say, instructing his followers to engage in battle against an enemy faction, the specifics of how the confrontation unfolds is handled on a per-unit basis by the game's AI routines. In sports simulations, 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. In adventure games, imitating human performance would first require an AI capable of visually recognising everyday objects and comprehending their significance, as well as an ability to understand and partake in conversations with other characters; this prerequisite level of common-sense reasoning makes the genre infeasible for imitation purposes, at least at present. 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 against those written by others.

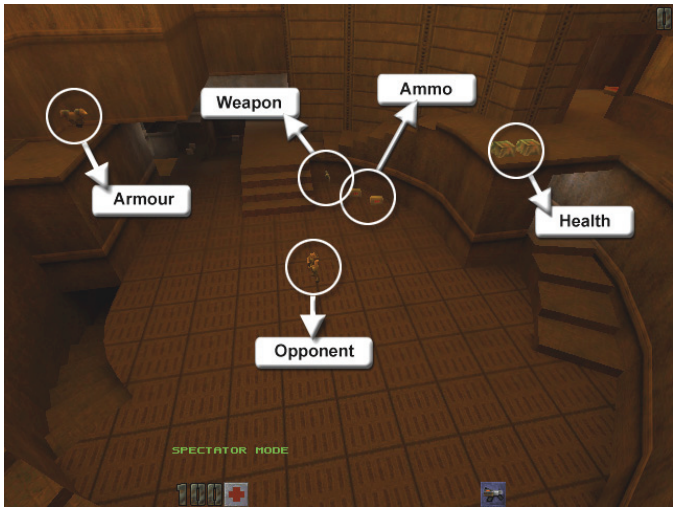

Figure 1 - Typical Quake 2 environment

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. While subsequent first-person shooter games have boasted faster gameply and more advanced visuals, the core features of the genre - those elements which are of particular interest to AI researchers, as outlined above - are as well-represented in Quake 2 as in its descendants. **Figure 1** shows a typical environment and features.

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 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 first-person games.

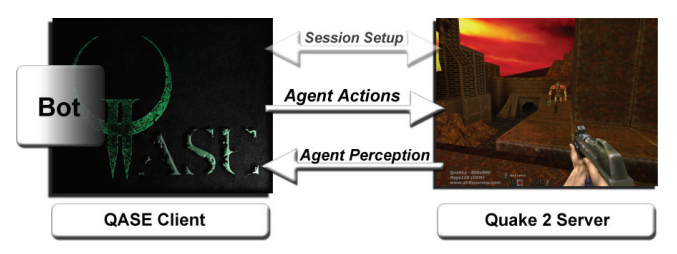

Figure 2 - The QASE API and its role in realising Quake agents

In the following sections we will outline the major components of the QASE architecture, highlighting its potential for application in both 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 data back to the server. Some complexity is introduced due to the fact that the client and server interpret positional and orientation data differently. From the server's perspective, the *forward* velocity corresponds to velocity along the global x-axis, *right* velocity is velocity along the global y-axis, and angular measurements are absolute. From the client's perspective, the *forward* velocity is velocity in the direction the agent is currently facing, the *right* velocity is perpendicular to this, and the angular measurements are relative to the agent's local axes. Thus, in order to realize artificial agents (also known as *bots*), a means of handling the game's network traffic and translating smoothly between *global* server data and *local* client data 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**. Synchronisation is employed to ensure the consistency of the gamestate across any parallel threads. All this is implemented transparently to the agent; 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.

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 on each frame, thereby building an inventory representation in real-time. 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.

Team-Based Play

QASE is fully compatible with the Threewave CTF modification for Quake 2, in which players join either a Red or Blue team and attempt to capture the enemy faction's flag. Methods are provided which enable the agent to join a specific team, or to join randomly; further methods allow the agent to determine whether a particular player is a member of its own or the opposing group. In cases where the server type is not known in advance, the API will automatically determine the game mode, and if necessary will join an arbitrary team. QASE is, therefore, well suited to researchers whose interest lies in investigating team-based behaviours and interactions.

Bot Hierarchy

While the network layer and gamestate-handling features described above are technically enough to facilitate the creation of in-game agents, they operate at too low a level to be practical for general use; they do not represent a rigorous, structured framework for the creation of Quake bots. Rather than requiring users to write agents from scratch and to manually handle the more menial aspects of client-server administration, QASE implements a structured *hierarchy* of bot classes, allowing rapid prototyping and development of agents from varying 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 a well-defined, standardised interface to which all agents must conform, but does not provide any further functionality; the programmer is entirely responsible for the actual implementation of the bot, 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

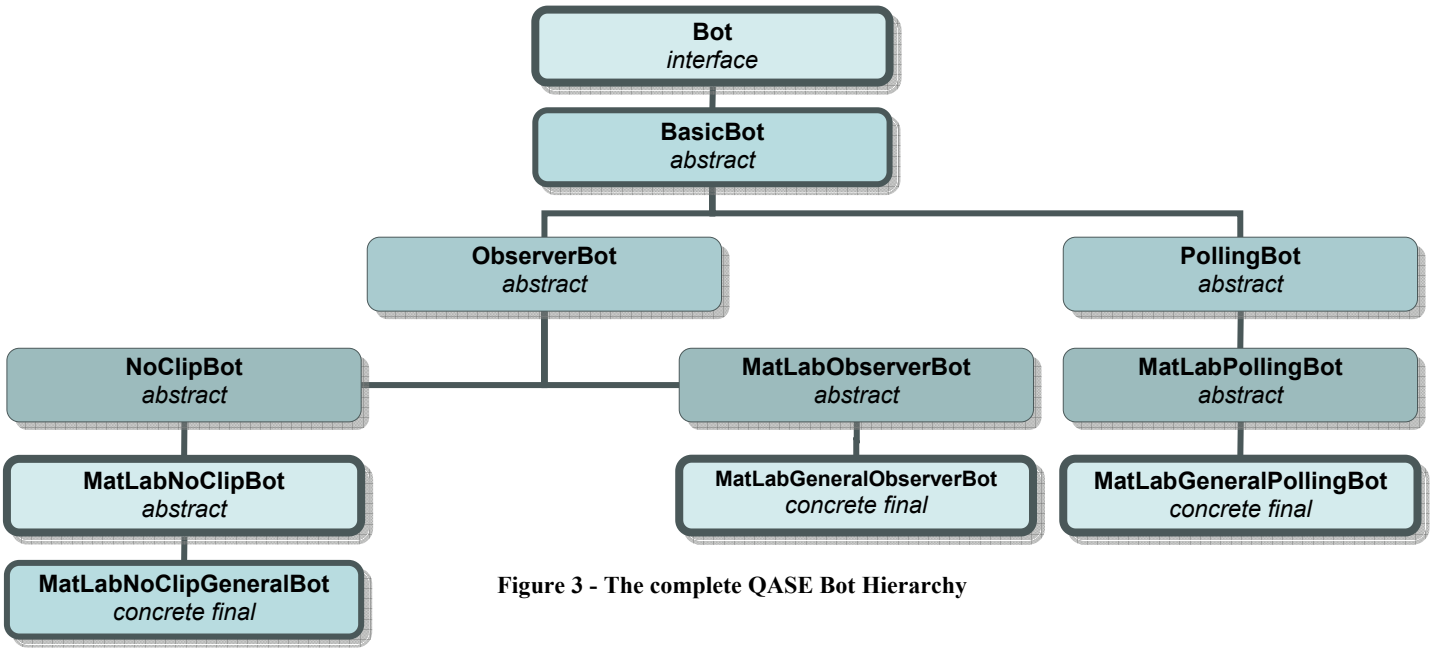

Figure 3 - The complete QASE Bot Hierarchy

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 (s)he wishes to use. The third level of the bot hierarchy provides ready-to-use implementations of two such paradigms (see below).

`BasicBot` also provides tailored, transparent access to the functions of the `BSPParser` class for environment sensing and the `WaypointMap` class for navigation (see later), by incorporating methods which relay calls to the appropriate embedded object. Users can also obtain a pointer to the underlying objects, thereby allowing full access to their functionality. Certain parameters are pre-defined to the most useful values; for instance, the bounding box used to trace through the level by the `BSPParser` is set to the size of the agent's in-game character's bounding box. `BasicBot` will also transparently find, load and query the BSP file associated with the current game level when one of the environment-sensing methods is invoked for the first time. Naturally, all these facilities are inherited by classes further down the bot hierarchy.

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 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. Thus, the agent is notified of each update as it occurs, a copy of the gamestate is presented to it, the user-defined AI routines set the required movement, aiming and action values for the next update, and the API auto-transmits the changes.

The *ObserverBot* uses the *observer* pattern to register its interest with the observable *Proxy*, and is thereafter notified whenever a game update takes place. Since this approach is single-threaded, a separate thread is created to check whether the bot has been killed, and to respawn as necessary. The advantages of this approach are twofold:

- it guarantees consistency of the gamestate; since the *Proxy* thread invokes a method call in the ObserverBot, it must wait until the agent's AI routine is complete before receiving any further updates.
- it allows multiple *observers* to connect to a single `Proxy`. This can be useful if the programmer wishes, for instance, to have a second observer perform some operation on the incoming data in the background.

The *PollingBot*, as its name suggests, operates by continually polling the *Proxy* and obtaining a copy of the gamestate *World* object. If a change in the current frame number is detected, the agent knows that an update has occurred, and will enter its AI routine. Because the *Proxy* and agent are operating on separate threads, the *Proxy* is free to receive updates regardless of what the agent is currently doing; this multithreaded approach may improve performance slightly, but could potentially result in changes to the gamestate arriving while the agent is executing its AI cycle, if said cycle is excessively long. To prevent this, the bot can optionally be set to *high thread safety mode*, in which the agent and *Proxy* both synchronize on the gamestate object; this means that the agent cannot read the gamestate while it is being written, and the *Proxy* cannot write the gamestate while it is being read.

Miscellaneous Bots

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, which we have found to be extremely useful in the course of our own research - indeed, the bot category was added specifically to address our need for such functionality. The *MatLabBot* branches facilitate integration with the MatLab programming environment, and will be explained later. The full hierarchy is shown in **Figure 3**.

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 are organised into blocks, each of which consists of a series of concatenated *messages* representing the network packet stream received by the client during the game session; the demo file therefore captures the player's every action and the state of all game entities at each discrete time step. For our own investigations in the field of imitation learning, a means of parsing these files and extracting the gameplay samples is essential. 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.

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. QASE accomplishes this by separating the header information received when entering each new level from the stream of standard packets received during the course of the game. 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** below shows the operation of each different trace type.

Environmental Entity Parsing

Aside from pure geometric data, the BSP files also contain information about certain active features within the game environment. These entities, which include doors, lifts, teleporters and buttons, should not be confused with the entity information received from the server on each update, which relates primarily to player movements and weapon spawns / despawns. The QASE API transparently parses and extracts the details of all such entities upon the first BSP query, and performs additional processing in order to allow the resulting data to be queried from high-level contexts. For instance, graph-style edge links are created between teleporters and their destination portals, while methods within the BasicBot class can be used to easily determine whether the player is currently standing on a moving platform.

Inbuilt AI Constructs

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. The main classes involved in this process are:

NeuralNet

Builds the network given design parameters, controls the retrieval and updating of its weights, facilitates output using *logsig* or *tansig* functions, and computes the net's output for given input. Also allows the network to be saved to disk and loaded at a later time.

Genetic

The genetic algorithm generator class, which maintains the gene pool, records fitness stats, controls mutation and recombination, and generates each successive generation when prompted. The class also provides methods to save and load Genetic objects, thereby allowing the genetic algorithm process to be resumed rather than restarted.

GANNManager

Provides the basic template of a 'bridge' between the GA and ANN classes, and demonstrates the steps required to evolve the weights of a population of networks by treating each weight as a nucleotide in the GA's genome. The class provides two modes of operation. For offline experiments - that is, those performed outside a live Quake 2 match - the GANNManager can be run as a thread, continually assessing the fitness of each network according to a user-defined function, recombining the associated genomes, and evolving towards an optimal solution for a specified duration of each generation and of overall simulation time. For online experiments, the class can be attached as an Observer of one or more Proxy objects, providing direct feedback from the Quake 2 game world. The class is abstract; it must be subclassed to provide the necessary fitness and observer functions, and to tailor its operation to the specific problem

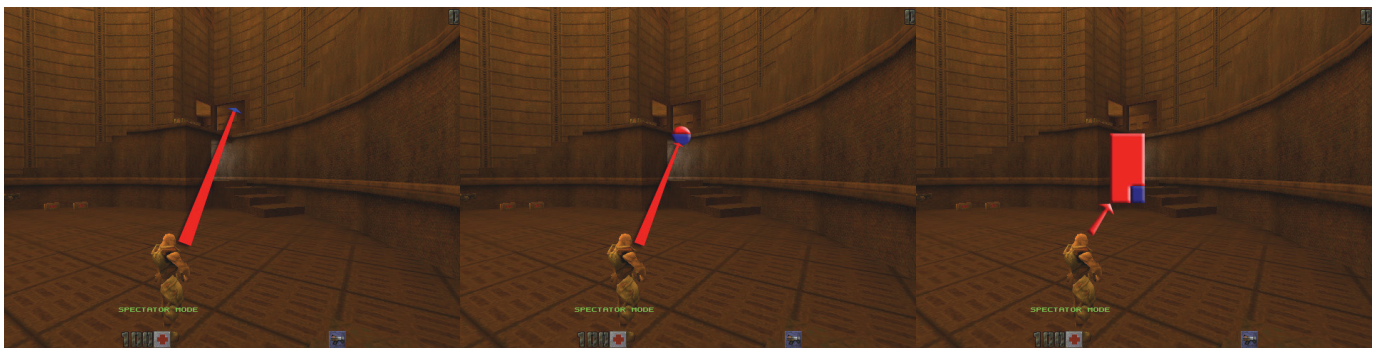

Figure 4 - BSP traces with line, sphere and box. Collision occurs at different points.

at hand. The class also allows the user to save an entire simulation to disk, and resume it from the same point later.

A *k-means* 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 intended 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.

Waypoint Maps

One of QASE's most useful features, particularly from an educational point of view, is the aforementioned *waypoint map generator*. The most important requirement of any agent is that it be capable of negotiating its environment. Although this can be done using the environment-sensing facilities outlined above, to rely exclusively upon BSP tracing would be a rather cumbersome and computationally expensive solution; most traditional methods of navigation instead employ *waypoint maps* - topological graphs of the level, indicating the paths along which the agent can move. With this in mind, QASE provides a package, `soc.ai.waypoint`, specifically designed to facilitate the rapid construction of such maps.

While the two principal classes of this package, Waypoint and WaypointMap, can be used to manually build a topology graph from scratch, QASE also offers a far more elegant and efficient approach to the problem - the `WaypointMapGenerator`. Drawing on concepts developed in the course of our work in imitation learning, this simply requires the user to supply a pre-recorded DM2 file; it will then automatically find the set of all positions occupied by the player during the game session, cluster them using the inbuilt k-means classes 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 need 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 bots require, and present the resulting demo file to QASE.

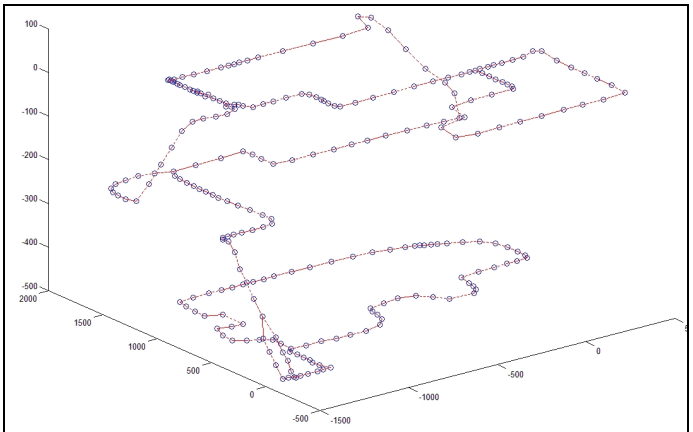

Figure 5 - Map of an in-game environment, created using WaypointMapGenerator, exported to and visualised in MatLab.

The waypoint map functionality is embedded into the BasicBot class; that is, it provides shortest-path methods which the agent transparently passes on to an underlying `WaypointMap` object. The ability to retrieve the map as raw positional and edge data is also provided; this is particularly convenient for reading the map into MatLab, as shown in **Figure 5**. Additionally, `WaypointMap` permits instances of itself to be saved to disk and reloaded, thereby enabling users to generate a map once and use it in all subsequent sessions rather than recreating it each time.

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® 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

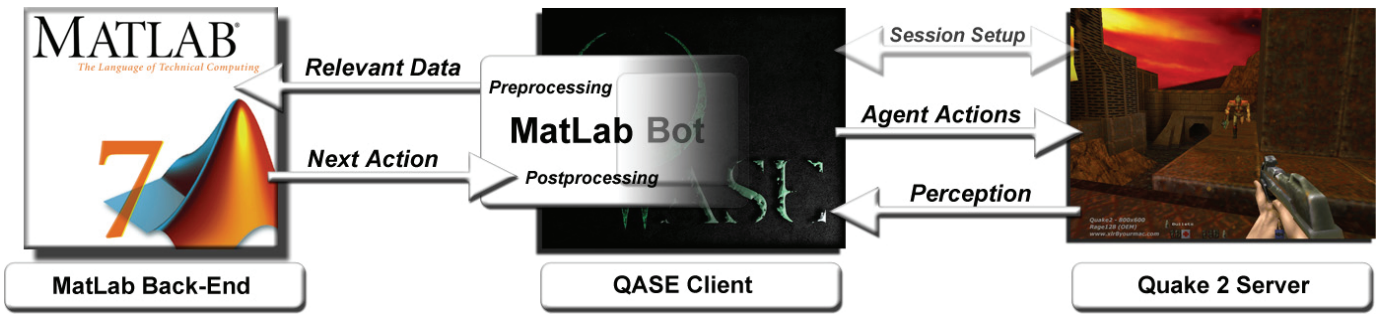

Figure 6 - MatLab/QASE integration. MatLab acts as a back-end in the AI cycle; the agent's body and brain are separated

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. We refer to agents created in this way as *direct* MatLab agents.

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 quite inefficient; for this reason, we developed an alternative paradigm designed to offer far better performance. 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's insertion into the AI cycle, a new branch of what we refer to as *hybrid* agents - the `MatLabBots` - was created. Each of these possesses a three-step AI routine:

1. On each server update, the custom QASE agent first *pre-processes* the data required for the task at hand; it then (automatically) 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 pre-processing 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 pre-processing 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 5** shows an example of this, using the `WaypointMapGenerator` in conjunction with `DM2Parser` to create a map of the game environment from human demonstration; see the section *QASE and Imitation Learning* for more.

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; QASE is designed for broad AI research, and the ability to build a back-end using MatLab - a tool with which researchers are already intimately familiar - is ideally suited to this purpose.

QASE AND OTHER APIs

During the initial exploratory phase of our research, we investigated a number of candidate APIs, originally intending to adopt one of them for use in our own experiments; as mentioned earlier, we were not satisfied that any of them provided the tools we would ultimately need. Here, we discuss these APIs. While each has some positive elements, we explain why in each case we felt that they fell short of our ideal platform, and in what ways we designed QASE to be a preferable alternative.

Quake 2 Bot Core
One of the earliest attempts to facilitate bot programming in Quake 2, the Bot Core (Swartzlander 1999) comprises an implementation of the game's client-side network protocol written in pure C, together with a very basic template for creating an AI cycle. It soon became apparent that it was a less than ideal platform for our work; parts of the network protocol had been neglected, other parts were not functioning reliably, it required that each agent be compiled into a separate executable, and its potential extensibility was extremely limited, making it an unsuitable choice for high-end research applications in general.

GameBots
The GameBots project (Adobbati et al 2001) allows communication between Epic Games' *Unreal Tournament* and other software, channelling messages to and from the server via a socket interface. Messages are both sent and delivered as ASCII strings, adhering to a predefined format. While this allows a wide range of programs to interface with the server, it is a bare-bones system; the user must write his own parser for the game messages, there are no supporting AI structures or logic included in the API, and communication is handled exclusively through scripting, with no low-level access available. Additionally, the socket interface which exposes the gamestate to external control is implemented via a modified version of the game server; it is therefore not possible to create an agent and connect it to an arbitrary Unreal Tournament match - agents can only communicate with a server running the GameBots mod.

Quagents
The Quagents project (Brown et al 2005) is intended primarily to propose Quake as a virtual testbed for robot and "ant colony"-style agent simulations. Its approach is quite similar to that of the GameBots API described above; Quagents is a recompiled modification of the Quake 2 libraries, which exposes the internal workings of the game to external manipulation via predefined script commands. Again like GameBots, the implementation of the actual agent controller itself is left to the user, although a sample application allowing for real-time control of the bot is supplied. However, Quagents also modifies the content of the game itself, implementing a stripped-down version of the original by eliminating many items, simplifying the agent's movement functions, and reducing the interactivity of the bot with the game world (Quagents, for instance, cannot engage in combat against each other).

In the case of both GameBots and Quagents APIs, their respective constraints - the elimination of features present in the original game, the limitations imposed by requiring the server to have the relevant modification installed, the lack of supporting structures and functionality - were among the primary reasons for our ultimate decision to reject them as viable candidate platforms for our reseach.

FEAR SDK

The most mature of the pre-existing first-person shooter APIs we investigated, FEAR (Champandard 2002) is once again a modification of the game's shared libraries; unlike GameBots and Quagents, however, FEAR also provides a framework for creating the agent controller itself. These must be compiled as DLLs and placed in appropriate subdirectories under the main FEAR mod folder; bots are then deployed by issuing commands from the server console during a game session. This has the drawback, however, of requiring not only that the server be running FEAR - as with GameBots and Quagents - but that the code for all desired agents be present on the same machine when the game session begins. If, for instance, multiple researches from different institutes wish to compare their agents, they cannot simply connect to a common server and deploy them. FEAR also includes a variety of in-built AI structures, including FSAs, decision trees, neural networks and rule-based systems. While such features are welcome, we felt that the inability (or at least, significant difficulty) of allowing external processing during the agent's AI cycle was quite limiting. MatLab, for instance, provides a vastly greater range of functionality than that embedded within FEAR, and is already a familiar working environment for many researchers. The SDK is also quite unintuitive in certain respects, which attenuates its utility as an educational aid. Moreover, the fact that the SDK is inextricably linked to the game engine renders it incompatible with other mods, such as the CTF team-based modification described earlier, unless the user were to manually subsume the CTF code into FEAR.

TIELT

The Testbed for Integrating and Evaluating Learning Techniques (Molineaux & Aha 2005) is a middleware platform designed to act as a generic intermediary between game engines and decision systems, somewhat similar to QASE's MatLab integration architecture as described earlier. For each game, users employ TIELT's inbuilt editor and scripting language to develop a series of knowledge bases; these consist of XML files defining elements such as the objects contained in the gamestate, events that may occur, state transition rules, messaging formats for communication with the game server and decision system, etc. The majority of applications have thus far centred upon real-time strategy games, although a bot has also been created for Unreal Tournament as proof-of-concept. While TIELT is an excellent general-purpose tool for research across different games, it is by necessity removed from the low-level details of each; we felt that our work - and that of other groups interested in pursuing research in first-person shooter games - would be better served by a consolidated API with a wide range of inbuilt functionality. Had we adopted TIELT for the purposes of interfacing our decision systems with Quake 2, we would have needed to write separate packages for reading human behaviour data from demo files, dealing with BSP geometry and environmental entities like lifts and doors, constructing navigation graphs, and so forth; this would have resulted in precisely the kind of ad-hoc, non-reusable amalgamation of software that we wished to avoid.

QASE: OUR APPROACH

Having examined the platforms outlined above, we concluded that none provided the combination of simplicity, power, modularity, reusability and extensibility that an educational and research tool of this kind required. We felt, as noted earlier, that the lack of such a platform was a major impediment to the adoption of first-person games in both areas, and sought to remedy this; it was always our main priority that the API be designed for the widest possible range of applications, rather than constraining itself to the specific needs of our own work. As such, QASE incorporates not only solutions to the various hurdles we encountered in the course of our research, but all the features which occurred to us as being potentially beneficial for others. Written in Java, the API itself consists of a single ~160kb JAR library, which can run unmodified on any JRE-enabled machine. As detailed elsewhere, it permits low-level access to gamestate and environmental information for those who want it, while also supplying high-level interfaces and convenience functions which perform the necessary gamestate-handling tasks transparently. It provides illustrative built-in AI structures for educational purposes, and facilities both the automatic generation and manual construction of full navigational systems. Its efficient and flexible MatLab integration provides an extremely powerful back-end engine with which researches are already intimately familiar. Unlike the approaches adopted by several of the APIs described above, its network layer encapsulates a full client-side implementation of the Quake 2 network protocol, meaning that it is cleanly decoupled from the server implementation; a QASE agent can connect to and be deployed upon any arbitrary Quake 2 server, Windows or Linux, modified or otherwise. It comes with full, detailed documentation and a series of articles focussing on the network protocol, the BSP file structure, and other subjects which will be of use to researchers embarking upon work in this area. Finally, it is worth noting that QASE is still in active development, and is evolving in response to the comments of groups and individuals who have adopted it; the other APIs mentioned above, with the exception of TIELT, are no longer maintained.

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. The following is drawn from our papers *"Towards Integrated Imitation of Strategic Planning and Motion Modelling in Interactive Computer Games"* (Gorman & Humphrys 2005) and *"Believability Testing and Bayesian Imitation in Interactive Computer Games"* (Gorman et al 2006).

One of the first questions which arises when considering the problem of imitation learning is, quite simply, "what behaviours does the demonstration encode?" A well-structured model of the human player's actions would facilitate an organised analysis of the observation data, greatly aiding the imitation process. 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 7**.

Strategic behaviours refer to actions the player takes with long-term goals in mind, such as adjusting his traversal of the map to maximise the number of items in his possession. *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).

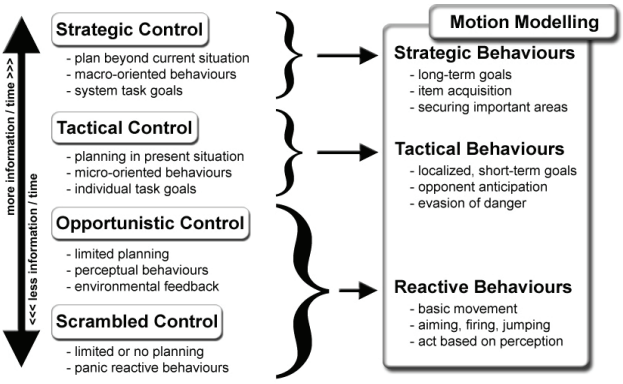

Figure 7 - Thurau's adaptation of Hollnagel's COCOM

Goal-Oriented Strategic Behaviour

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*. In Quake 2, experienced players traverse the environment methodically, controlling important areas of the map and collecting items to strengthen their character. Thus, we define the player's long-term goals to be the *items* scattered at fixed points around each level. By learning the mappings between the player's status and his subsequent item pickups, the agent can adopt observed strategies when appropriate, and *adapt* to situations which the player did not face.

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* - the list of what quantities of which items he currently possesses - using the inventory-tracking facilities of *DM2Parser* described earlier, and we then obtain the set of *unique* inventory states; these *inventory prototypes* represent the varying situations

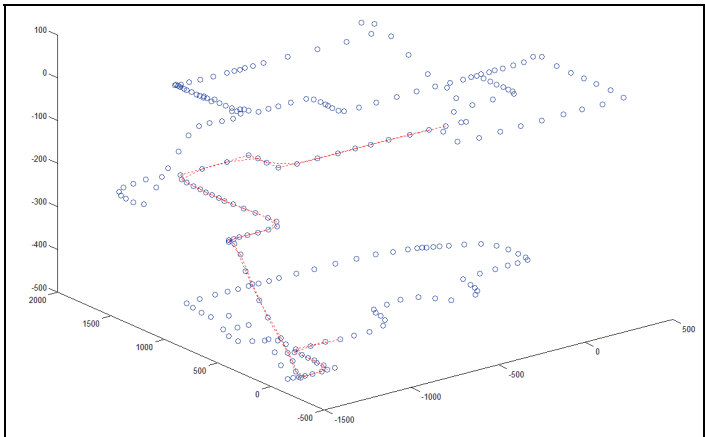
Figure 8 - An example of a path followed by the player while in a particular inventory state. The path originates in the lower part of the level, and ends at the point where the player picked up an item that caused his inventory to shift towards another prototype.

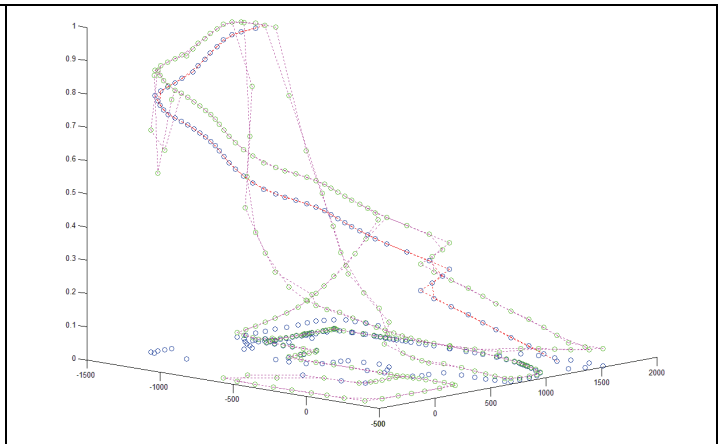
Figure 9 - The ascending rewards assigned to one of the paths followed by the player (blue/red), and the results of the value iteration algorithm (green & magenta). The y-axis deontes the values associated with each waypoint in the topological map.

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 this.

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

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

where p_i is a prototype, and $c_{i,j}$ is the j^{th} 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.

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.

Bayesian Motion Modelling

In the course of a game session, human players exhibit actions other than simply moving along the environment surface, including jumps, weapon changes and discharges, crouches, etc. In many cases, the player can only attain certain goals by performing one or more such actions at the appropriate time; they therefore have an important *functional* aspect. From the perspective of creating a believable agent,

it is also vital to reproduce the human *aesthetic* qualities they encode - the agent should not, for instance, be capable of instantaneously turning 180°, since this would be impossible for the human's mouse-and-keyboard interface. For the purposes of our agent, then, a means of imitating these actions is essential.

In a previous contribution, Thurau et al describe an approach based on Rao, Shon & Meltzoff's Bayesian inverse-model for action selection in infants and robots. The choice of action at each timestep is expressed as a probability function of the subject's current position state ct, next position state c_{t+1} and goal state c_g, as follows:

$$P(a_t \mid c_t, c_{t+1}, c_g) = \frac{P(c_{t+1} \mid c_t, a_t) P(a_t \mid c_t, c_g)}{\sum_u P(c_{t+1} \mid c_t, a_u) P(a_u \mid c_t, c_g)}$$

It is immediately clear that this model fits into the strategic navigation system almost perfectly; the clusters c_t and c_{t+1} are chosen by examining the utility values, while the current goal state is implicitly defined by the membership distribution. In order to derive the probabilities, we read the sequence of actions taken by the player as a set of vectors v such that

$$v = [\Delta yaw, \Delta pitch, jump, weapon, firing]$$

We then cluster these action vectors to obtain a set of action primitives, each of which amalgamates a number of similar actions performed at different times into a single unit of behavior.

Several important adaptations must be made in order to use this model in the game environment. Firstly, in practice we decouple the yaw and pitch elements of the action vector from the remainder, and sequence them separately - this produces a more fine-grained clustering of the primitives. Secondly, Rao's model assumes that transitions between states are instantaneous, whereas multiple actions may be performed in Quake 2 while moving between successive clusters; we therefore express $P(c_{t+1}|c_t,a_t)$ as a soft-distribution of all observed actions on edge $E_{ct,ct+1}$ in the topological map. Third, Rao assumes a single unambiguous goal, whereas we deal with multiple weighted goals in parallel. We thus perform a similar weighting of the probabilities across all active goal clusters. Finally, Rao's model assumes that each action is independent of the previous action. In Quake 2, however, each action is constrained by that performed on the preceding timestep; we therefore introduce an additional dependency in our calculations. The final probabilities are computed as follows:

$$\sum_g m_g P(a_t \mid c_t, c_{t+1}, c_g) \frac{P(a_t \mid a_{t-1})}{\sum_u P(a_u \mid a_{u-1})}$$

The priors can now be derived by direct examination of the observation data.

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`, implementing `preMatLab` such that it

Figure 10 - Examples of a QASE agent in action, drawn from our experiments in imitation learning. The top sequence shows the agent *leaning into* and *strafing around* a corner, as a human player does. In the middle, the agent's next goal is an item on top of the box. As it approaches, it looks downwards, jumps, and fires a rocket to propel itself upwards. This so-called *rocket jump* is considered an advanced move and is commonly employed by players to reach otherwise inaccessible areas. Bottom, the agent interacts with a lift by standing still as it ascends, then jumps off at the top, an unnecessary action which is nonetheless common among human players.

extracts the player's current position and inventory from the gamestate; these are then passed to MatLab. We also rewrite the MatLab template script to instantiate the agent and connect it to the server. On each update, MatLab determines the closest matching state prototype and node, extracts and weights the relevant utility configurations, finds the set of nodes connected to the current node by examining the edge matrix, and selects the successor with the highest utility value; it then examines the current position cluster, the cluster to which the agent is moving, the current goal distribution, and the last executed action primitive, computing from this the appropriate subsequent pitch, yaw, weapon, jump and firing state. All this data is then passed back to QASE and is received by the agent's *postMatLab* method, which we have implemented such that it computes the direction between its current position and the next node and sets the agent's movement accordingly; the bot's orientation is simultaneously altered to reflect the specified action primitive. 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 yet intuitive API as a major impediment to the adoption of commercial games in AI education and research. We then presented our QASE API, which was developed to meet these requirements. Several of its more important features were described, and their usefulness highlighted; these features were then compared against existing game-based artificial intelligence APIs. A practical demonstration of QASE as it has been used in our own research closed this contribution.

Since its release, QASE has already attracted attention from several quarters. From our correspondence, we know that it is currently used at Bielefeld University, Germany; it has been adopted by researchers at China's Huazhong University and at Deustche Telekom; and it is used as both a research tool and undergraduate teaching aid at the University of Las Palmas de Gran Canaria. The number of downloads recorded thus far, along with some casual Google searches, suggest that several other groups are also utilising it in their work. As more such individuals discover QASE, the resulting feedback will aid us in continually improving the API; as part of that continuing effort, 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.

FUTURE WORK

Although we regard it as being very feature-rich 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.

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

REFERENCES

Adobbati R, Marshall AN, Scholer A, Tejada S, Kaminka G, Schaffer S, and Sollitto C (2001): "Gamebots: A 3D virtual world test-bed for multi-agent research", in Proc Second International Workshop on Infrastructure for Agents, MAS, and Scalable MAS

Bauckhage C, Thurau C. & Sagerer G. (2003): Learning Humanlike Opponent Behaviour for Interactive Computer Games, Pattern Recognition, Vol 2781

Brown CM, Ferguson G, Barnum P, Bo Hu, Costello D (2005): "Quagents: A Game Platform for Intelligent Agents", in Proc First Artificial Intelligence and Digital Entertainment Conference

Byrne, R.W. and Russon, A.E. "Learning by Imitation: A Hierarchical Approach", Behavioral and Brain Sciences (1998) 21, 667-721

Champandard AJ (2002): FEAR SDK, http://fear.sourceforge.net

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

Molineaux M & Aha D(2005): "TIELT: A testbed for gaming environments". Proceedings of the 16th National Conference on Artificial Intelligence (pp. 1690-1691)

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

Swartzlander B (1999): Quake 2 Bot Core, http://www.telefragged.com/Q2BotCore

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

B. Gorman, C. Thurau, C. Bauckhage, and M. Humphrys: Believability Testing and Bayesian Imitation in Interactive Computer Games, In Proc. 9th Int. Conf. on the Simulation of Adaptive Behavior (SAB'06), LNAI 4095, p 655-666

APPENDIX A: QASE PACKAGE STRUCTURE

Having detailed the intent, design and implementation of the API, we here provide QASE's full package structure. This will help to situate the concepts discussed above, and is intended as a practical aid for programmers wishing to avail of the features offered by QASE. For further details, please consult the QASE Specification and the Javadoc accompanying the API.

QASE (`soc.qase`) consists of several packages. While perhaps somewhat daunting at first glance, the overall structure of the API is designed to be as intuitive and modular as possible. The following section describes the function of each package in greater depth.

soc.qase.ai

This package contains a number of subpackages and classes designed to provide some in-built AI functionality. Intended primarily for education purposes, since more heavy-duty research work can make use of the MatLab integration discussed earlier.

soc.qase.ai.gann

Consists of classes which implement a GANN architecture - that is, neural networks which learn through the application of evolutionary algorithms. Designed to allow students to examine and experiment with ready-made implementations of these common undergraduate constructs.

soc.qase.ai.kmeans

The `kmeans` package is designed to give students an insight into some basic principles of clustering techniques, and how they can be used. It is also heavily employed by various classes in the `waypoint` package (see below).

soc.qase.ai.waypoint

Facilitates the creation of navigation systems for the agent. Of particular note is the `WaypointMapGenerator` class, which takes a recording of a human player traversing a level and automatically builds a full navigation system using the `kmeans` and `dm2` packages, according to concepts outlined in (Gorman & Humphrys 2005). Waypoint maps can also be built manually. See the relevant section in the main paper.

soc.qase.bot

Contains classes facilitating the creation and control of Quake 2 agents from differing levels of abstraction. See Appendix B for more details.

soc.qase.bot.matlab, soc.qase.bot.matlab.general

soc.qase.bot.matlab.sample

Contain classes which enable the integration of QASE with the MatLab programming environment. See the relevant section of the main text for more.

soc.qase.bot.sample

Contains a number of sample agents designed to demonstrate the procedure involved in writing an autonomous bot.

soc.qase.com

The classes contained in the `com` package are used to implement low-level communication between a proxy component and the Quake 2 server. Mostly, therefore, these objects are for internal use, and can be ignored by a casual user of the API. However, the `com` package also includes the physical implementation of the QASE agent interface, in the form of the `Proxy` class.

soc.qase.com.packet

Consists of classes representing each packet type used in the course of a game session (client-to-server, server-to-client,

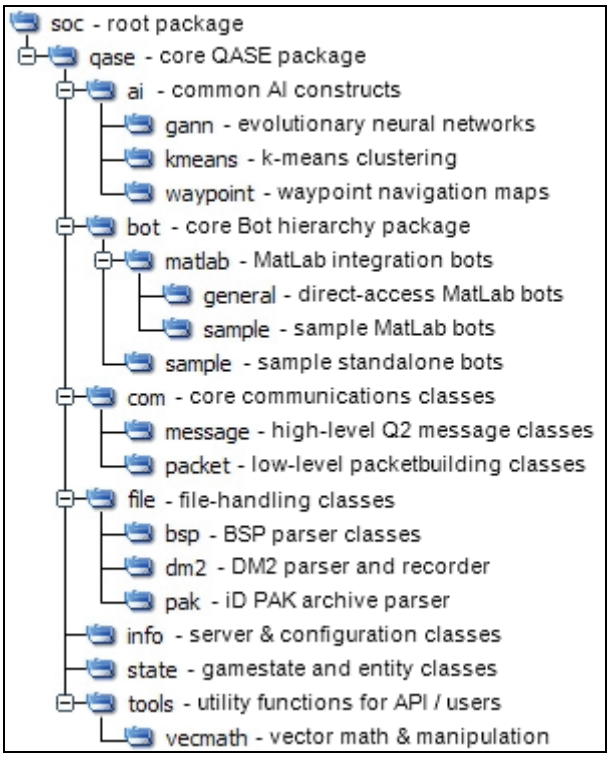

***Figure 11** - Package outline of the QASE API*

connectionless). Used as wrappers for the various `Message` classes, whose type and content are derived from the packet payloads.

soc.qase.com.message

Contains classes which encapsulate the data conveyed by each type of message used in a game session, including both client-to-server messages (move agent, etc) and server-to-client messages (gamestate info, configuration, etc)

soc.qase.file

Consists of a set of subpackages designed to allow QASE to parse different file formats used by Quake 2, to store resource archives, recorded game sessions, or world geometry.

soc.qase.file.bsp

Contains the `BSPParser` and related classes. This allows QASE to read, parse and query the geometry of a game level, which Quake 2 stores in local BSP (Binary Space Partition) files. The `BasicBot` class contains a number of environment-sensing methods which automatically find, load and query the current in-game map, using `BSPParser` in conjunction with `PAKParser`.

soc.qase.file.dm2

Contains the `DM2Parser` and `DM2Recorder` classes. The former allows QASE to read the recorded demo of a match by treating the .dm2 file as a virtual server, 'connecting' to it and reading the gamestate at each frame as it would during an online session. The latter allows QASE agents to automatically record themselves to file during play; this actually improves upon the standard Quake 2 recording facilities, by allowing matches spanning more than a single map to be recorded in playable format.

soc.qase.com.pak

The PAKParser class allows QASE to read and extract the contents of a PAK, the format use by Quake 2 to store multiple resource files in a single consolidated archive. Used extensively by `BasicBot` to automatically find and load the current map.

soc.qase.info

There are a number of classes in the `info` package, mainly intended for internal use by the API itself; these relate to the transfer of configuration data between server and client. The only class that should actually be used directly by the programmer is the `User` class which specifies player options, and even this is automated by the existing `Bot` hierarchy.

soc.qase.state

Contains classes representing each of the elements defining the current state of the game world and the desired change-of-state effected by the agent. The former includes game entities, the agent's movement, its status, inventory and gun, irregular events and sounds; the latter includes the agent's velocity, orientation and actions.

soc.qase.tools

Contains miscellaneous tools used throughout the API. The core `Utils` class provides methods to generate random numbers, convert byte arrays to data types and vice-versa, convert angular measurements to 2D vectors, parser environment variables, and more.

soc.qase.com.vecmath

Contains `Vector2f` and `Vector3f` classes which facilitate 2D and 3D vector manipulation.

APPENDIX B:
CREATION OF QASE AGENTS

In this section, the procedures involved in creating different types of QASE agent are outlined by demonstration. QASE agents fall into one of two broad categories; *standalone* agents, in which the AI routine is completely atomic and internalised within the runAI method inherited from `BasicBot`, and *MatLab* agents, wherein QASE acts as a "gamestate filter" and the AI routines themselves are implemented in the MatLab environment.

Standalone Agents

Creating a standalone QASE agent is extremely straightforward, thanks to the degree of automation provided by the API. All the programmer needs to do is create a subclass of `ObserverBot`, `PollingBot` or `NoClipBot` as appropriate, and write the necessary AI routines by implementing the abstract runAI method. Examples of both Polling and Observer agents are included with QASE, in the `soc.qase.bot.sample` package; each of these bots, when connected, will simply run directly towards the closest item in the game environment, collect it, and move on.

MatLab Agents

QASE agents which use MatLab as a back-end engine fall into one of two subcategories - they can either be *direct* MatLab agents, or *hybrid* agents. The former involves using one of the `MatLabGeneralBots` and writing the entire AI routine, including all gamestate parsing operations, within a MatLab script; this is the most straightforward approach, but is also quite computationally costly. The latter involves creating a subclass of `MatLabObserver` / `Polling` / `NoClipBot`, filtering the gamestate on each update, and passing only the minimal necessary state information to a partner MatLab script; this has the dual advantage of being more efficient and of allowing both script and agent to be modular and reusable. *Hybrid* agents, due to their significantly better performance, are the preferred paradigm - all our own research is conducted using hybrids. For both agent categories, QASE automates all information-passing functions, requiring only that the programmer write the AI routines themselves. Template MatLab scripts are supplied.

Before any agents can be created, however, it is necessary to import the API into the MatLab environment. The MatLab scripts supplied with the QASE API include `prepQASE.m`, which automates this process. All that is required is for the

user to edit the script to reflect the location at which the library JAR file is stored on his/her machine.

```
% substitute path to QASE lib on current system
javaaddpath('C:\path_to_QASE\qaselib.jar');

import soc.qase.com.*;
import soc.qase.info.*;
import soc.qase.state.*;
import soc.qase.bot.matlab.sample.*;
import soc.qase.bot.matlab.general.*
import soc.qase.file.dm2.*;

% add any further imports here
```

Figure 12 - ***prepQASE.m*** *imports common packages into the MatLab environment*

Direct MatLab Agents

Creating a *direct* agent involves simply editing the supplied BotManagerGeneralTemplate.m script, to supply the gamestate-parsing and AI routines in the main loop. When creating direct agents, it is preferable to use `MatLabGeneralPollingBot` rather than `MatLabGeneralObserverBot`, as the former gives superior performance; no such performance discrepancy exists in the case of hybrid agents. An example of one such agent is provided with the API. As with the standalone agents, the `SampleBotManagerGeneral.m` script will connect a bot to a local server, and instruct it to continually pursue the nearest active item.

Hybrid MatLab Agents

The advantage of using *direct* agents is that the MatLab code is very simple, and closely resembles that of a standalone agent. However, because it requires MatLab to perform a significant amount of Java object manipulation, it becomes quite computationally inefficient if large quantities of data are needed from the gamestate. Additionally, it means that a new script must be written from scratch for each agent. With this in mind, we developed an alternative paradigm designed to fulfil two criteria:

- maximise efficiency by *minimising cross-platform communication* between MatLab and the JVM
- separate the ***body*** of the agent (QASE) from its ***brain*** (MatLab), allowing both to be modular and reusable

As mentioned in the "MatLab Integration" section earlier, the standalone bots' AI routines are internalised and atomic, contained entirely within the runAI method. In order to facilitate MatLab, a new branch of agents, the *MatLabBots*, was created. Each of these conceptually possesses a three-step AI routine as follows:

1. ***Pre-process*** the data required for the task in QASE
2. ***Processes*** the input data natively in MatLab according to specified AI routines
3. ***Post-process*** the output data in QASE, applying it to the agent as appropriate.

The *pre-processing* step filters the gamestate, presenting only the minimal required information to MatLab and thereby enabling both MatLab and Java to process as much data as possible in their respective native environments. In practice, QASE automates all transfer of data between QASE and MatLab, requiring only that the programmer supply the actual AI routines themselves. The steps involved in creating a hybrid agent are as follows:

1. Create a concrete subclass of *MatLabObserver / Polling / NoClipBot*. In particular, implement the two abstract methods as follows:

- **`preMatLab`** - extract the input required for the given task from the gamestate and format it according to how MatLab will subsequently use it, placing the data into the supplied **`Object[]`**. Typically, this array will be populated with a series of individual **`float[]`** arrays, each supplying a different element of the input.

- **`postMatLab`** - apply the results obtained from MatLab's AI routines as appropriate. Again, this output usually takes the form of an **`Object[]`** populated with individual **`float[]`** arrays.

2. Edit the template MatLab script in **`BotManagerTemplate.m`** to supply the necessary AI routines, performing computations upon the input data and placing the results in the output cell array provided. No direct action need be taken to affect the agent's state; this will be done passively when the output data is passed to QASE.

The framework for passing data to and from MatLab is automated in QASE's *`MatLabBots`*, and the abstract methods and scripts are designed in such a way as to minimise the amount of code the programmer must write. By separating the concerns of each platform in this manner, we furthermore facilitate the reuse of both scripts and derived agents across different experiments.

APPENDIX C:
QASE AGENTS - CODE SAMPLES

Over the following pages, we present some code samples to better illustrate the creation of QASE agents. The examples below are drawn from the sample agents included in the API itself (for standalone QASE bots) and the MatLab template scripts which accompany it (for direct / hybrid MatLab bots).

```
        public void runAI(World w)
        {
                ...

                world = w;
                player = world.getPlayer();
                entities = world.getItems();

                ...

                // find nearest item entity
                for(int i = 0; i < entities.size(); i++)
                {
                        tempEntity = (Entity)entities.elementAt(i);

                        tempOrigin = tempEntity.getOrigin();
                        entPos.set(tempOrigin.getX(), tempOrigin.getY(), 0);

                        tempOrigin = player.getPlayerMove().getOrigin();
                        pos.set(tempOrigin.getX(), tempOrigin.getY(), 0);

                        entDir.sub(entPos, pos);

                        if((nearestEntity == null || entDir.length() < entDist)
                        && entDir.length() > 0)
                        {
                                nearestEntity = tempEntity;
                                entDist = entDir.length();
                        }
                }

                // set subsequent movement in direction of nearest item
                if(nearestEntity != null)
                {
                        tempOrigin = nearestEntity.getOrigin();
                        entPos.set(tempOrigin.getX(), tempOrigin.getY(), 0);

                        tempOrigin = player.getPlayerMove().getOrigin();
                        pos.set(tempOrigin.getX(), tempOrigin.getY(), 0);

                        entDir.sub(entPos, pos);
                        entDir.normalize();

                        setBotMovement(entDir, null, 200, 0); // set movement dir
                }
        }
```

Figure 13 *- Extract from* ***SampleObserverBot.java****, an example of a standalone QASE agent (abridged)*

```
function [] = BotManagerGeneralTemplate(botTime, recordFile)
    prepQASE;   % import the QASE library

    try
        % create and connect the bot
        matLabBot = MatLabGeneralPollingBot('MatLabGeneralPolling','female/athena');
        matLabBot.connect('127.0.0.1',-1,recordFile);

        tic;

        % loop for the specified amount of time
        while(toc < botTime)
            if(matLabBot.waitingForMatLab == 1)
                % World state read from agent %
                % app-dependent computations here %
                % fov, velocity, etc applied directly %
                matLabBot.releaseFromMatLab;
            end

            pause(0.01);
        end
    catch
        disp 'An error occurred. Disconnecting bots...';
    end

    matLabBot.disconnect;
end
```

Figure 14 - *The* ***BotManagerGeneralTemplate.m*** *script file.* ***Direct*** *MatLab agents are created by editing this template to add the required AI computations in the main loop, as indicated.*

```
function [] = SampleBotManagerGeneral(botTime, recordFile)
    prepQASE;   % import the QASE library

    try
        matLabBot = MatLabGeneralPollingBot('MatLabGeneralPolling','female/athena');
        matLabBot.connect('127.0.0.1',-1,recordFile);

        pos = [];
        entPos = [];
        entDir = [];
        entDirVect = soc.qase.tools.vecmath.Vector3f(0,0,0);

        tic;

        while(toc < botTime)
            if(matLabBot.waitingForMatLab == 1)
                world = matLabBot.getWorld;

                tempEntity = [];
                nearestEntity = [];
                nearestEntityIndex = -1;
                entDist = 1e10;

                tempOrigin = [];

                player = world.getPlayer;
                entities = world.getItems;
                messages = world.getMessages;

                matLabBot.setAction(0, 0, 0);

                for j = 0 : entities.size - 1
                    tempEntity = entities.elementAt(j);

                    tempOrigin = tempEntity.getOrigin;
                    entPos = [tempOrigin.getX ; tempOrigin.getY];

                    tempOrigin = player.getPlayerMove.getOrigin;
                    pos = [tempOrigin.getX ; tempOrigin.getY];

                    entDir = entPos - pos;

                    if((j == 0 | norm(entDir) < entDist) & norm(entDir) > 0)
                        nearestEntityIndex = j;
                        entDist = norm(entDir);
                    end
                end

                if(nearestEntityIndex ~= -1)
                    nearestEntity = entities.elementAt(nearestEntityIndex);

                    tempOrigin = nearestEntity.getOrigin;
                    entPos = [tempOrigin.getX ; tempOrigin.getY];

                    tempOrigin = player.getPlayerMove.getOrigin;
                    pos = [tempOrigin.getX ; tempOrigin.getY];

                    entDir = entPos - pos;
                    entDir = normc(entDir);

                    entDirVect.set(entDir(1, 1), entDir(2,1), 0);

                    matLabBot.setBotMovement(entDirVect, entDirVect, 200, 0);
                end

                matLabBot.releaseFromMatLab;
            end

            pause(0.01);
        end
    catch
        disp 'An error occurred. Disconnecting bots...';
    end

    matLabBot.disconnect;
end
```

Figure 15 - SampleBotManagerGeneral.m. *This* ***direct*** *MatLab agent closely parallels the* ***SampleObserverBot.java*** *source shown above. MatLab is responsible for obtaining the gamestate (****matLabBot.getWorld****), querying it to extract the required information, performing the necessary computations, and manually setting the agent's subsequent actions (****setBotMovement****).*

```
% botTime specifies the bot's lifetime in seconds
function [] = BotManagerTemplate(botTime, recordFile)
    prepQASE;   % import the QASE library

    mlResults = cell(1, 1); % allocate a cell array to contain MatLab's results

    try
        % create and connect the bot - can be either built-in or custom
        matLabBot = SampleMatLabObserverBot('MatLabObserver','female/athena');
        matLabBot.connect('127.0.0.1',-1,recordFile);

        tic;

        % loop for the specified amount of time
        while(toc < botTime)
            if(matLabBot.waitingForMatLab == 1)
                mlParams = matLabBot.getMatLabParams;

                % app-dependent computations here %
                % place results in mlResults cell array %

                matLabBot.setMatLabResults(mlResults);

                matLabBot.releaseFromMatLab;
            end

            pause(0.01);
        end
    catch
        disp 'An error occurred. Disconnecting bots...';
    end
```

Figure 16 - *The* ***BotManagerTemplate.m hybrid agent*** *script. Users need only specify the size of the* ***mlResults*** *cell array (ie number of outputs) and the app-dependent computations as indicated.*

```
function [] = SampleBotManager(botTime, recordFile)

    prepQASE;   % import the QASE library

    mlResults = cell(1, 1);

    try
        matLabBot = SampleMatLabObserverBot('MatLabObserver','female/athena');
        matLabBot.connect('127.0.0.1',-1,recordFile);

        tic;

        while(toc < botTime)
            if(matLabBot.waitingForMatLab == 1)
                mlParams = matLabBot.getMatLabParams;

                pos = mlParams(1);
                entPos = mlParams(2);
                dir = normc(entPos - pos);

                mlResults{1} = dir;

                matLabBot.setMatLabResults(mlResults);

                matLabBot.releaseFromMatLab;
            end

            pause(0.01);
        end
    catch
        disp 'An error occurred. Disconnecting bots...';
    end

    matLabBot.disconnect;
end
```

Figure 17 - *The* ***SampleBotManager.m*** *script, an example of a* ***hybrid agent*** *script. Input consists of the position of the agent and that of the nearest item, each in the form of a float array. Output is the direction the agent needs to move, in the form of a MatLab cell array (which is auto-converted into an array of Java objects). As can be seen, separating Java and MatLab processing using hybrid agents results in clear, efficient and highly intelligible code.*