

The QASE Specification 2.0

Original by

Martin Fredriksson [martin.fredriksson@bth.se] ¹

Version 2.0 Updated & Written by

Bernard Gorman [bgorman@computing.dcu.ie] ²

¹ Dept. of Software Engineering and Computer Science,
Blekinge Institute of Technology,
Ronneby, Sweden.

² School of Computing,
Dublin City University,
Glasnevin, Dublin 9,
Ireland

Contents

1.	INTRODUCTION	3
1.1	MOTIVATION	3
1.2	SIMULATORS AND RELATED CONCEPTS.....	4
1.3	QASE APPLICATION PROGRAMMING INTERFACE	5
1.4	OUTLINE OF THE SPECIFICATION.....	6
2.	QASE / QUAKE 2 ENVIRONMENT	7
2.4	QUAKE 2 OVERVIEW	7
2.5	MAPS	8
2.6	FIXED ENTITIES	8
2.7	MOVING ENTITIES.....	9
2.8	RESPAWNING	9
2.9	SETTING UP A SIMULATION	9
3.	QASE AGENT INTERFACE.....	10
3.1	QUAKE 2 NETWORK PROTOCOL	10
4.1	FRAMES OF EXECUTION.....	12
3.2	AGENT INTERFACE / PROXY RELATIONSHIP	12
4.	QASE API STRUCTURE.....	15
4.1	PACKAGE STRUCTURE OUTLINE	15
4.2	PACKAGE DETAIL.....	15
5.	ADVANCED QASE FEATURES.....	18
5.1	BOT HIERARCHY	18
5.2	GAMESTATE AUGMENTATION	19
5.3	DM2PARSER AND DM2RECORDER	20
5.4	ENVIRONMENT SENSING.....	20
5.5	MATLAB INTEGRATION.....	23
5.6	WAYPOINT MAP	24
5.7	INBUILT AI FUNCTIONALITY	25
6.	CREATING QASE AGENTS.....	27
6.1	STANDALONE QASE AGENTS	27
6.2	QASE MATLAB AGENTS	28
	REFERENCES	32
	APPENDIX: FIXED ENTITIES	33

1. Introduction

"Agents are autonomous, computational entities that can be viewed as perceiving their environment through sensors and acting upon their environment through effectors. To say that they are autonomous means that to some extent they have control over their behavior and can act without intervention of humans and other systems. Agents pursue goals and carry out tasks in order to meet their design objectives, and in general these goals and tasks can be supplementary as well as conflicting." [1]

1.1 Motivation

1.1.1 Cognitive Agents and Artificial Intelligence

During the last decade or so, the research area of artificial intelligence has received new attention as the result of a growing interest in agent systems and multiagent systems. One of the reasons for this new interest in artificial intelligence is probably due to the commonly-accepted definition of agent properties (see extract above), defining an agent as an autonomous entity engaged in some kind of interaction with an environment. The strong relationship between agents and artificial intelligence manifests itself in the attribute of autonomy - how can we design and implement a software entity that is able to act on its own in some environment?

The question above mainly involves the investigation of three components: an environment, a number of agents, and the interfaces used by the agents to interact with the environment. However, not unlike other areas of research, in order to pursue the investigation of these components and in what way they relate to each other we need a problem domain where the components can be appropriately applied and further examined.

In order to fulfill the above task, we propose a problem domain involving a physical environment with tangible entities and objects. Furthermore, the environment should also imply one or more well-defined goals that the agents interacting with the environment can strive to fulfill. In our proposed environment, there exists only one quantifiable overall goal; eliminate as many adverse agents as possible during a fixed amount of time.

1.1.2 Commercial Computer Games

Despite a history of games-based research, including Tesauro's TD-Gammon [7] and IBM's infamous Deep Blue, academia has generally regarded commercial games as a distraction from the serious business of AI, rather than as an opportunity to leverage this existing domain to the advancement of our knowledge [6]. Part of this attitude has been caused by the type of games traditionally used; while chess and its ilk provide a useful testbed for the development of efficient searching and pruning techniques, they present problems to which computers are inherently suited, and do not offer the potential that modern commercial computer games do. Indeed, some now argue that work in this area has become so constrained that it does a disservice to the promotion of games-based research in general [6]. Additionally, such games bear little or no relevance to the real world, and it is thus difficult to see how they can contribute significantly to an intelligence capable of the broad range and scope of human behaviour. Games used in a research capacity are also typically custom-written for the intended purposes (e.g. Sklar [13]), which makes it difficult to compare results with even those researchers using similar custom environments; by contrast, commercial games represent objective, universal standards, written simply to provide players with a challenging experience rather than with research in mind. One of the recent pioneers of academic games AI, John Laird, argues that the field at large has become so fragmented, concerned with the development of ever more specialised algorithms designed to work in increasingly narrow problem domains, that the "holy grail" - the development of an integrated

human-level intelligence - has been all but forgotten [9]. It is his contention, given the rich complexity of interactive computer game environments and the fact that they are inherently designed to present problems whose solutions require human reasoning abilities, that such games represent the single best platform upon which to tie the various strands of AI research into a cohesive whole and construct new models of artificial intelligence.

1.2 Simulators and Related Concepts

In order to design and implement an agent that pursues some kind of task in a physical environment, we generally have to design and implement the environment in question. Since there already are such environments implemented in the form of advanced computer games, our approach will be to use one such game as a simulator platform and then implement the agents and their interfaces to the environment separately. The specific game referred to is **Quake 2** [2].

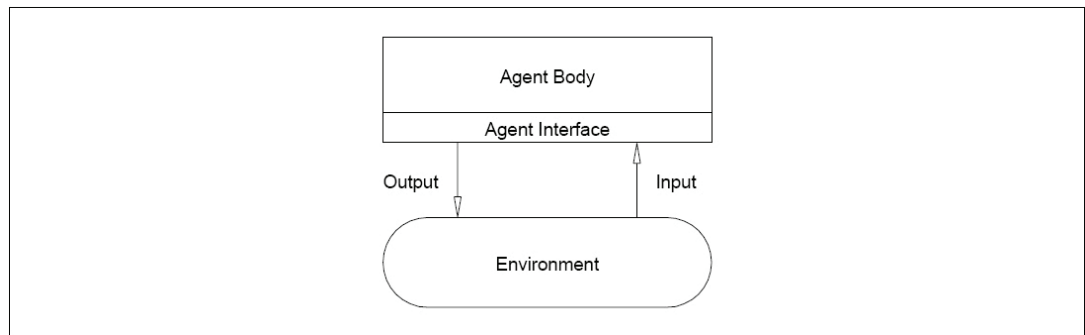
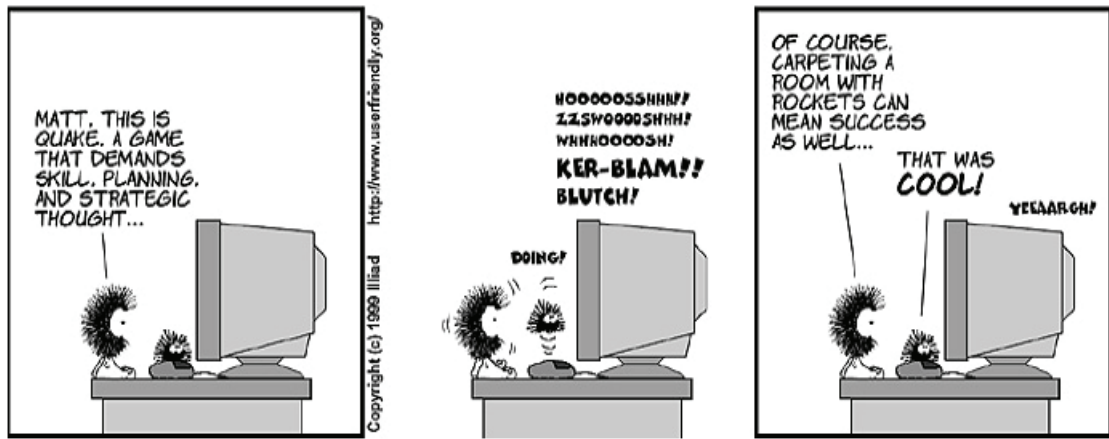


Figure 1 - Main components of the simulator: agent, interface, and environment

Basically, Quake 2 is a computer game divided into two components: a client component and a server component. The client component is used to visualize a game taking place in a three-dimensional space, as well as taking care of input generated by some controlling party. The input is forwarded to a server component that uses the input to change the current state of an environment, that the client (and possibly a number of other clients) interact with. The server continuously sends information concerning the current environment state to all connected clients for the duration of the game session. Further details are provided in Chapters 2 and 3.

Quake 2 was adopted for a number of reasons. Firstly, it was prominent in the literature, having been chosen by Laird for his experiments using the Soar architecture [11][12]. Second, it is a mature game, and as such the existing resources are somewhat more substantial than for other games. Thirdly, rather than focussing attention on a single element of human behaviour, Quake 2 matches require a full spectrum of ‘multiplexed’ reactive, strategic and tactical behaviours from the agent, executed in competition against opponents of equal skill; from the perspective of research, this offers a huge number of potential avenues for investigation. Thirdly, the first-person shooter genre - while incorporating unrealistic features such as invincibility, teleportation, etc - is nonetheless an abstraction of the real world, as opposed to a board or puzzle game with no relevance to reality. Finally, Quake 2 provides facilities for the playback of recorded matches, allowing us to incorporate features into our API which enable agents to record their game sessions directly to disk for later viewing and analysis.



Environment. The environment previously mentioned is handled by the Quake 2 server component, and is basically constituted by three-dimensional layouts (called maps), and tangible entities, manifested as something from here on referred to as *fixed* and *moving* entities. The agents interacting with the environment are an example of such moving entities. The simulated environment's state can be both accessed (read) and affected by an agent; these two operations are performed via an agent interface.

Interface. The agent interface depicted in **Figure 1** is used by an agent to interact with the simulated environment. There are three major classes of interaction modes implemented by an agent's interface: connectivity handling (connecting to and disconnecting from an environment), reading environment state, and writing environment state.

Agent. The objective of Quake 2 - and consequently for the agents referred to in this specification - is to eliminate as many agents as possible while staying alive as long as possible. However, fulfilling this goal requires both skill and cunning, and implementing a computational entity that exhibits such behavior is quite a challenge both from the perspective of autonomous agents and the perspective of artificial intelligence. As an aid in solving this task, and to facilitate analysis of one specific class of autonomous agents (inhabiting a simulated physical environment), we propose using Quake 2 in combination with externally-implemented agents. The state information that is possible to extract from the environment via the agent interface is of such magnitude that a large number of autonomous agent and classic artificial intelligence issues can be effectively investigated.

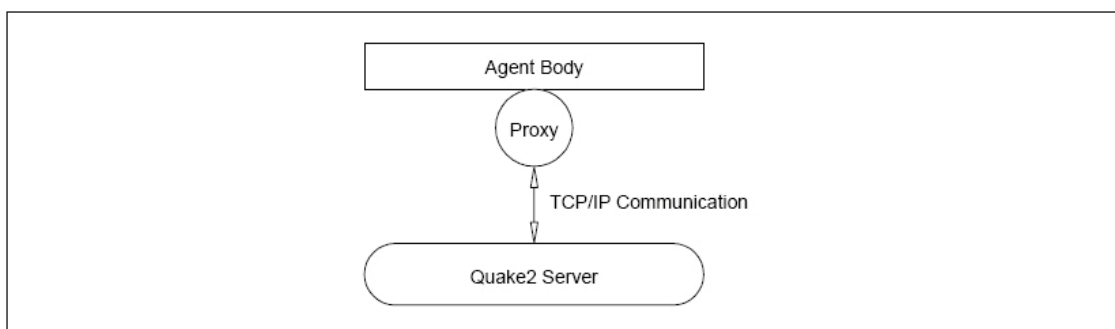


Figure 2 - Actual implementation of the suggested simulator design. The previously mentioned agent interface corresponds to a proxy object handling all communication (synchronous as well as asynchronous) between the agent and the environment.

1.3 QASE Application Programming Interface

Our approach of using an already existing environment simulator is tightly coupled with the idea of implementing external agent bodies and associated interfaces. Therefore, it is important that the agents and their interfaces are easy to implement, or even better, if the functionality they imply is already implemented and available on most hardware

platforms. In order to fulfill this goal, we have developed a comprehensive, feature-rich Java API, called the Quake 2 Agent Simulation Environment (QASE); it is intended to provide all the functionality necessary 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. By presenting this API to the research and educational community, we hope to foster further interest in the adoption of computer games in academic AI.

1.4 Outline of the Specification

This first introductory chapter of the QASE specification explained the motivation behind this work, our decision to use existing commercial computer games as a simulation environment, our choice of Quake 2 as an ideal testbed, and the various concepts involved in creating an agent-environment interface. We then proposed our QASE API as a platform for research in artificial intelligence and agent systems using commercial computer games. The rest of this specification is divided into more extensive descriptions and discussions of these issues. Chapter Two delves into the Quake 2 environment and related concepts (three-dimensional layout schemes, fixed entities, moving entities, etc). In Chapter Three, the interface used by the agent to interact with its environment will be further discussed, along with an explanation of the Quake 2 network protocol. Chapter Four concerns the package structure of QASE, briefly outlining the purpose of each. Chapter Five introduces more sophisticated QASE features, providing an overview of the major functionality offered by the API. Finally, Chapter Six provides detailed instructions on how to write standalone and MatLab-augmented agents using QASE.

A paper on the API, “*QASE - An Integrated API for Imitation and General AI Research in Commercial Computer Games*”, was accepted by and published in the proceedings of the 7th International CGAMES Conference in November 2005 [5].

2. QASE / Quake 2 Environment

As previously mentioned, the simulator described in this specification involves agents, their interfaces, and an environment that the agents inhabit. There can be any number of environment layouts present, but a simulation can only be carried out using one specific environment at a time. However, the different environments do have a number of common characteristics: they make use of a three-dimensional layout scheme (called a map), they are populated with a number of fixed entities, and they can possibly also be populated with a number of agents (human or software) engaged in combat.

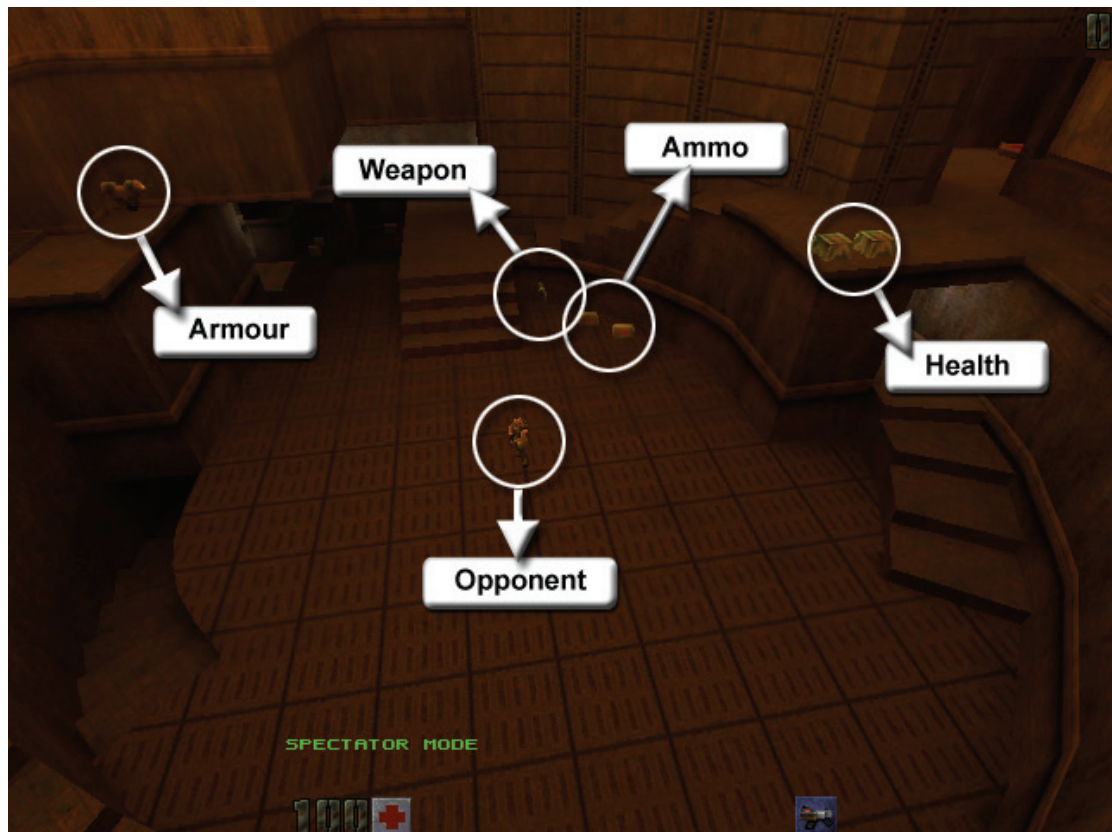


Figure 3 - Typical Quake 2 game environment, with various items annotated

2.4 Quake 2 Overview

Quake 2, first released by ID Software in 1997, belongs to the genre of *first-person shooters*. Players explore a three-dimensional environment littered with weapons, bonus items, traps and pitfalls, with the objective of reaching a particular endpoint and/or defeating as many opponents as possible. Most interesting from the perspective of autonomous agents is the multi-player mode, where players compete against one another in specially-designed *deathmatch* arenas.

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. These opponents then fight until a specified time or kill limit has been reached, at which point the game restarts on the same map or another, as dictated by the server. Thus, in order to realize artificial *bots* (agents), a means of establishing a session with the Quake 2 server, handling incoming gamestate information and communicating the bot's actions back to the server must be implemented.

2.5 Maps

Volume Boxes and Free Space. The environment maps are defined by a number of volume boxes (called *brushes*), rendered as physical obstacles for an agent present in the environment. An agent can move around the unoccupied space enclosed by these brushes, as in **Figure 4**.

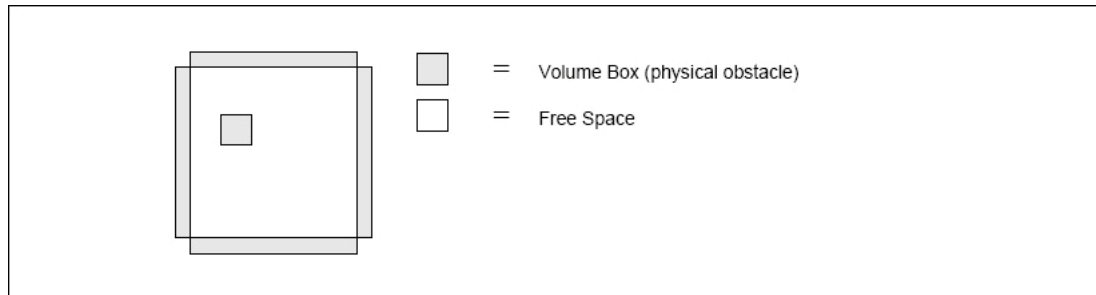


Figure 4 - Fundamental layout of an environment, involving five volume boxes (called ‘brushes’). An agent can move freely in the space enclosed by the four walls, apart from that occupied by the solid cube. For more information on brushes and their use in collision detection, see Section 5.4.

Coordinate System. The environment maps and their constituents (volume boxes, fixed entities and agents) are all mapped onto a three-dimensional orthogonal coordinate system.

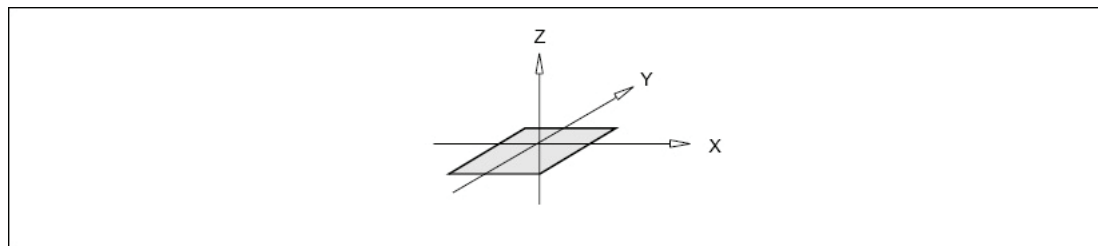


Figure 5 - The three-dimensional coordinate system. Note that *Quake 2* uses a rotation whereby the vertical axis is designated ‘z’ from the perspective of the agent.

2.6 Fixed Entities

In the environment, there can also be items denoted as ‘fixed entities’ present. These entities can be picked up by an agent and used in a variety of situations, helping the agent to survive against hostile opponents. The fixed entities can be divided into five major categories: armor, health, weapons, ammunition, and special items (see Appendix A: Fixed Entities for more).

Armor. There are three armor types: flak jacket, combat suit, and body armor. Each one provides a certain amount of protection against both normal attacks and energy weapon attacks. Armor strength is depleted by each hit.

Health. There are two types of standard health kits: first aid and medkits.

Weapons. There are a variety of weapons available: blaster, shotgun, super shotgun, machine gun, chaingun, grenades, grenade launcher, rocket launcher, hyper blaster, BFG10K, and rail-gun. Each has a particular advantage given the combat situation.

Ammunition. There are five major ammunition types: shells, cells, bullets, grenades, and rockets. Except for the blaster, you must have ammunition to use a weapon. Each ammunition type has a maximum which the agent can carry.

Special Items. There are nine different special items that can be present in a simulated environment: energy armor, bandoleer, heavy pack, underwater breather, environment suit, quad damage, mega health, invulnerability, and super adrenaline. These special items should be

used in different situations, ranging from protecting an agent from environmental hazards to making more damage to opponents, and carrying more ammunition, etc.

2.7 Moving Entities

As previously mentioned, an environment is constituted by a map populated with a number of fixed entities, but the environment can possibly also be populated with moving entities, i.e. agents. The agents can be of two different types - human or software. However, from an agent's perspective, it is not possible to make a distinction between these two types of moving entities; agent perceive all moving entities as fellow agents.

From an external perspective an agent can be characterized by a number of attributes, with some of these attributes uniquely identifying each individual agent.

Name. The current name chosen by the agent (or its designer).

Skin. The current skin model chosen by the agent (or its designer), representing a certain rendering model to be used if visualization of the environment is desired.

Location. This attribute represents the current location of an agent in the environment, and it is expressed in terms of X, Y, and Z coordinates.

Angles. This attribute represents the current angles of an agent, and is expressed in terms of x, y and z angles around the agent's own origin.

Velocity. The velocity attribute represents the current velocity of an agent in three directions: along the X axis, along the Y axis, and along the Z axis.

2.8 Respawnning

There is a phenomenon associated with the simulated environment and the entities inhabiting it that is never found in a real-world environment. In Quake 2, the phenomenon is referred to as *respawning*, or sometimes just *spawning*. The concept of respawning involves three elements: *entities* (fixed and moving), *map locations* (within free space), and *time*.

The basic idea behind respawning is that, if a fixed entity is picked up by an agent, it should reappear at the same location after a certain period of time. The specific location is an item's associated respawn position. The specific period of time mentioned is an item's associated respawn time. The concept of respawning can also apply to agents; if an agent is eliminated, it will respawn at one of several respawn positions (as chosen by the simulator, i.e. the Quake 2 server) as soon as the agent itself chooses to re-join the game, or the simulator forces it to.

2.9 Setting up a Simulation

Setting up the simulation environment mainly involves two steps: running the simulation server, and connecting a spectator or participant. The server is responsible for conducting the actual simulation, and the spectator is used to visualize the environment and chase specific players during a simulation. If you want to join an ongoing simulation, not as a spectator but rather as an in-game opponent, you must join the simulation as a non-spectator client. Below, you will find examples of all three start-up sequences.

```
SERVER:      Quake 2.exe +game dm +map <map> +set dedicated 1
SPECTATOR:   Quake 2.exe +connect <host>:27910 +spectator 1
OPPONENT:    Quake 2.exe +connect <host>:27910
```

3. QASE Agent Interface

The QASE agent interface refers to the interface used by an agent to communicate and interact with the simulated environment. Conceptually, the interface is constructed using input and output methods connected with the environment. However, from an implementation perspective, the interface provides three major classes of functionality: environment connectivity, reading environment state, and writing environment state.

3.1 Quake 2 Network Protocol

The most fundamental requirement of the API, of course, is that it be capable of connecting to and communicating with a game server. Thus, before we describe the specifics of the API, it is necessary to understand something of the Quake 2 network protocol itself.

3.1.1 Session Setup

Each game session is established using connectionless packets, a special type of packet which can be sent to the server in the absence of a proper connection. These packets consists simply of a header containing the sequence number -1, followed by a text string indicating the desired command. The session setup procedure is as follow:

- The client issues a packet to the server containing the string **getchallenge**
- The server replies with a packet containing the string

```
o challenge 12345
```

where 12345 is a code used to ensure that the client is not using a spoofed IP

- The client responds with

```
o connect 34 789 12345 <userdata>
```

where 34 is the network protocol revision, 789 is a unique client ID, 12345 is the code sent by the server earlier, and <userdata> is a text string containing information about the player's desired character model, name, etc.

- The client then receives a **ServerData** message providing global information about the game world and entities, as well as a series of **SpawnBaseline** messages which provide details of the various entities' initial (baseline) attributes.
- Finally, the client receives a **StuffText** message indicating that it is ready to enter the game world. One last connectionless packet is sent to the server:

```
o begin <levelkey>
```

where `levelkey` is an identifier for the current map, received in **ServerData**. Once all this is complete, the character is *spawned* into the world and can now interact with the other entities using the in-game protocol described below.

Connectionless packets can also be used to send irregular queries and commands to the server, as distinct from the standard movement and aiming information transmitted on every update. These include requests for a listing of the client's inventory, to talk to other players, to use inventory items, or to switch weapons. The QASE bot hierarchy (see later) provides convenient methods to use all these features, hiding the actual message construction from the user; direct calls to the `sendCommand` methods of the `Proxy` class are also facilitated.

3.1.2 In-Game Protocol

Once the session has been established and the agent has entered the game, the server and client begin exchanging messages. Every tenth of a second, the server sends updates to each connected client, providing information on the position, velocity, orientation, etc. of both the client itself and of other entities in the game world. Unless exceptional circumstances are encountered, these updates consists of two sequence numbers and three *messages*:

- **ServerFrame**, which specifies the current and delta reference frames,
- **PlayerInfo**, giving the client's current position, velocity, aim, weapon, score, etc.
- **PacketEntities**, which contains similar information about all other game entities.

Upon receiving each frame, the client must merge the updated information into its existing gamestate record (see below). This done, the client issues a ClientMove message to the server, indicating its desired velocity (*forward* $\in [-400, 400]$, *right*, *up* $\in [-200, 200]$), aim (*yaw*, *pitch* $\in [-180, 180]$), and whether or not it is firing its gun. These values together define the range of legal in-game movement.

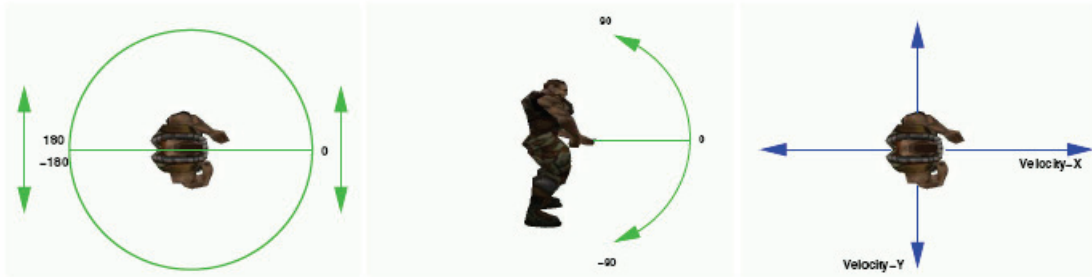


Figure 6 - Server's view of the in-game character's range of actions [8]

An important point to note is that the client and server interpret these values 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 towards which the agent is currently facing*, the *right velocity* is *perpendicular to this*, and the *angular* measurements are *relative to the agent's local axes*. **Figure 6** shows the server's perspective of the character, **Figure 7** shows the client's. If using data drawn from recorded demos, then, the API must be capable of moving back and forth between *local* bot co-ordinates and *global* server co-ordinates.

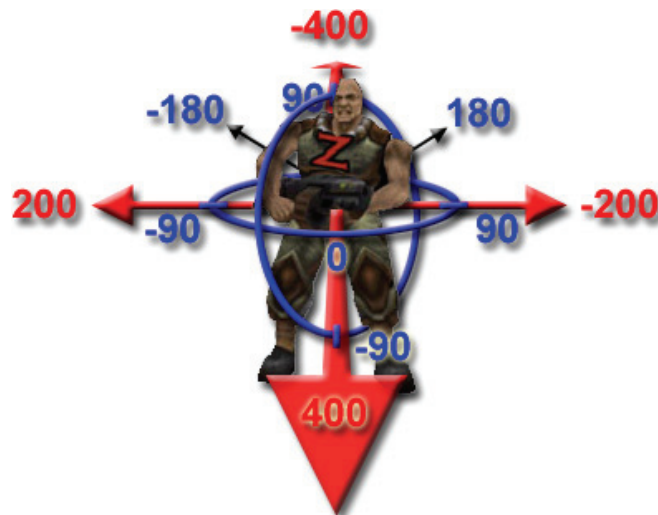


Figure 7 - Client's view of the in-game character's range of actions

3.1.3 *Delta Coding and Gamestate Consistency*

Since Quake is a real-time game in which potentially several dozen participants need to be updated every hundred milliseconds, it is important that network latency be minimised. As such, the UDP protocol is used for communication in preference to TCP; this means that packets may arrive out of order, or be lost altogether. This approach is not as reckless as it may first appear - the loss of a player update message (for instance) can be compensated in subsequent updates, whereas using TCP would cause any lost packets to hold up the data stream while the protocol waited for resent information which is, by that point, out of date. Given a suitably low-overhead mechanism for acknowledging and ordering incoming packets, the integrity of the gamestate can be ensured while simultaneously maximising the available bandwidth.

Quake 2 implements such a scheme by means of *sequence numbers* and *delta coding*. Both the client and server store old frame updates to allow for packet loss; the server stores the previous 12 frames, while the client stores the previous 16. To reduce bandwidth, the server only transmits information on entities within a certain viewable distance of the client, and will only send the specific entity attributes which have changed since the last frame received by the client. Each update packet sent from client to server has two sequence numbers as its header; the first specifies the number of the current frame, while the second specifies the number of the last frame received from the server. Using this information, the server can determine which packets were lost, and what cumulative changes have occurred in the game world since that point. On the next update, the server will transmit those changes, along with the reference frame against which the client should *delta code*; the client then *merges* the updates into its stored copy of the reference frame to produce a complete gamestate representation. If the number of lost packets exceeds the number of stored frames, then the server enters a special mode where it transmits entity updates based on the *baseline* (initial) values established during the client's connection. It will continue to do so until the client indicates that it has re-synchronized itself and is ready to resume delta coding.

4.1 **Frames of Execution**

A central concept when it comes to a QASE agent's behavior are *frames of execution*. A frame is an amount of time during which the agent performs a set of actions as a result of previously-received information concerning its environment and internal state. Each frame exists in the interval between server updates - that is, 100 milliseconds. During this time, the agent must decide upon which actions it wishes to execute over the course of the following frame, and transmit them to the server; in other words, the agent may perform a different action ten times each second. If the agent fails to transmit its desired actions in this interval, the server will assume that the last received actions are to be repeated during the new frame.

3.2 **Agent Interface / Proxy Relationship**

As previously mentioned, there is a slight difference between the conceptual definition of an agent's interface and the actual implementation of it. From a conceptual perspective, we denote the interface as a QASE agent interface, implementing two methods: state input and state change output. However, from the perspective of an actual implementation, the QASE agent interface is implemented as a software component called a `Proxy`. The concept of a proxy can be compared to that of a translator. From the server's point of view, the proxy component corresponds to an agent, and from the agents point of view the proxy component corresponds to an environment. In other words, the proxy component implements the functionality needed to tie an agent and its corresponding environment together, enabling them to communicate with and access each other.

The following sections provide a brief overview of the most important methods involved in connecting to, reading from and writing to the simulation environment. **It should be noted that QASE automates all such operations, allowing the programmer to concentrate on implementing the desired AI routines using QASE's more sophisticated facilities; a**

comprehensive look at these issues is offered in the “Advanced QASE Features” and “Creating QASE Agents” chapters. For low-level details of the methods and classes mentioned here, please consult the accompanying Javadoc.

3.1.4 Environment Connectivity

The first class of functionality that an agent interface should implement is the physical connection between an agent and the simulated environment in question (using a proxy component, this corresponds to basic socket communication). The connectivity functionality is divided into two methods: **connect** and **disconnect**.

- `Proxy.connect(String host, int port, String recfile)`
This method requires the host name of a Quake 2 server and the port number of this specific host, as well as a filename to which the game session should be recorded (or null for none). The method blocks until either the host name was not found, the host's port number was invalid, or a successful connection was established.
- `Proxy.disconnect()`
Used to close an established connection between the interface and environment.

3.1.5 Environment State Fundamentals

Once a successful connection has been established by the interface (i.e. proxy) between an agent and its corresponding environment, it is possible to extract state information from the environment. This includes both information about physical phenomena associated with the environment and the entities that inhabit it, but also visual rendering information concerning the current environment state. The visual rendering information is supplied by a Quake 2 server in order to enable a corresponding client component to correctly visualize on-screen information, in case the client component is controlled by a human user. This is usually not of interest to a completely autonomous agent interacting with a simulated environment, but the QASE application programming interface still includes both classes of information, just in case the designer of an agent would like to actually visualize an agent's perspective on the current environment state.

3.1.6 Reading Environment State

After a successful connection has been established by a proxy component between an agent and its associated environment, state information about the environment can be extracted. This information corresponds to the information received as input by an agent interface from an associated environment. The method used by the proxy component to read the current game environment is **getWorld**.

- `Proxy.getWorld()`
A method call to this function of a proxy component returns an instance of type `World`, which acts as a wrapper for all environment state information. The `World` object contains information about the agent itself, all visible entities, and messages sent to the environment by any inhabiting agent.

3.1.7 Extracting Gamestate Information

The `World` object mentioned above consists of a hierarchy of component objects representing the current gamestate. The class implements a number of methods designed to allow the programmer to easily access the constituent elements of the current environment state. The most important of these are **getPlayer** and **getEntities**.

- Proxy.getPlayer()
An important part of the current environment state is information concerning the agent itself, i.e. introspective attributes. Calling the `getPlayer` method will return a wrapper object of type `Player`, which in turn contains objects of type `PlayerMove`, `PlayerView`, `PlayerGun` and `PlayerStatus`, each encapsulating different elements of the agent's internal state
- Proxy.getEntities(...)
Returns a `Vector` containing `Entity` objects representing the various items, weapons, and other agents present in the environment. In a similar manner to the `Player` object, `Entity` is a wrapper for a visible entity and its associated state. Several different overloaded versions of this method exist, allowing the user to filter and customize the list of entities returned by it.

3.1.8 Writing Environment State

The second class of environment access methods that an agent interface should implement is the write environment state operators. In the actual implementation of a proxy this is handled by two methods: **sendMovement** and **sendCommand**.

- Proxy.sendMovement(Angles, Velocity, Action)
An agent issues a call to this method if it desires to change its current state - that is, its orientation, velocity, or action. All of these parameters must be passed when `sendMovement` is invoked. If no call to `sendMovement` is made, the proxy will send the last known movement state to the environment at each interval.
- Proxy.sendCommand(String command)
Using this method, the agent can change the current state of certain attributes which are separated from those agent attributes affected by calling `sendMovement`. These include the agent's current weapon, and the usage of any items it may be carrying.

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.

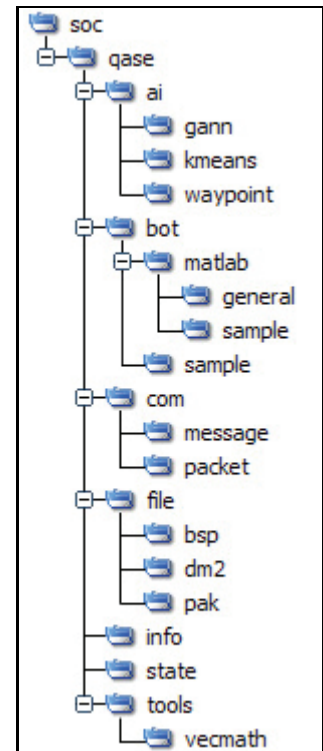
4. QASE API Structure

Having outlined the fundamental concepts underpinning the API and briefly described the core network functionality, we now describe the full package structure of QASE. This will help to situate the concepts discussed above, and will also aid in understanding the advanced features introduced in the following chapter.

4.1 Package Structure Outline

QASE (`soc.qase`) consists of several packages, each of which contains several subpackages of their own. 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.

Figure 8 (right) - Package outline of the QASE API



4.2 Package Detail

4.2.1 *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 below.

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) [4]. Waypoint maps can also be built manually. See the relevant section below.

4.2.2 *soc.qase.bot*

Contains classes facilitating the creation and control of Quake 2 agents from differing levels of abstraction. See “Creating QASE Agents” for more details.

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

Contains classes which enable the integration of QASE with the MatLab programming environment. See the relevant section of the “Creating QASE Agents” chapter for more.

soc.qase.bot.sample

Contains a number of sample agents designed to demonstrate the procedure involved in writing an autonomous bot. These are used as the examples in the “Creating QASE Agents” chapter below.

4.2.3 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 previously-mentioned QASE agent interface, in the form of a class called `Proxy`.

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, 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)

4.2.4 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 consolidate archive. Used extensively by `BasicBot` to automatically find and load the current game map.

4.2.5 soc.qase.info

There are a number of classes included 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 (see later).

4.2.6 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.

4.2.7 *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.

To clarify some of the concepts discussed in this outline of the package structure, the following chapter contains a higher-level discussion of the API's advanced features. For more low-level detail, please see the accompanying Javadoc documentation.

5. Advanced QASE Features

In this chapter, we present an overview of some of QASE's main features. These include the Bot Hierarchy (which supercedes and automates the connection and gamestate management methods outlined above, allowing the programmer to concentrate on implementing AI routines), integration with the MatLab programming environment, waypoint maps, environment sensing and more. A familiarity with the features in this chapter is essential before moving on to Chapter 6, "Creating QASE Agents".

5.1 Bot Hierarchy

The connection and gamestate-management methods described earlier operate at too low a level to be practical for general use; they do not represent a genuine, structured framework for the creation of game agents. Rather than requiring users to write bots from scratch and to manually handle menial tasks such as defining the bot's profile, respawning when it has been killed, etc, we decided to create a comprehensive *bot hierarchy* allowing users to develop agents from any of 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.

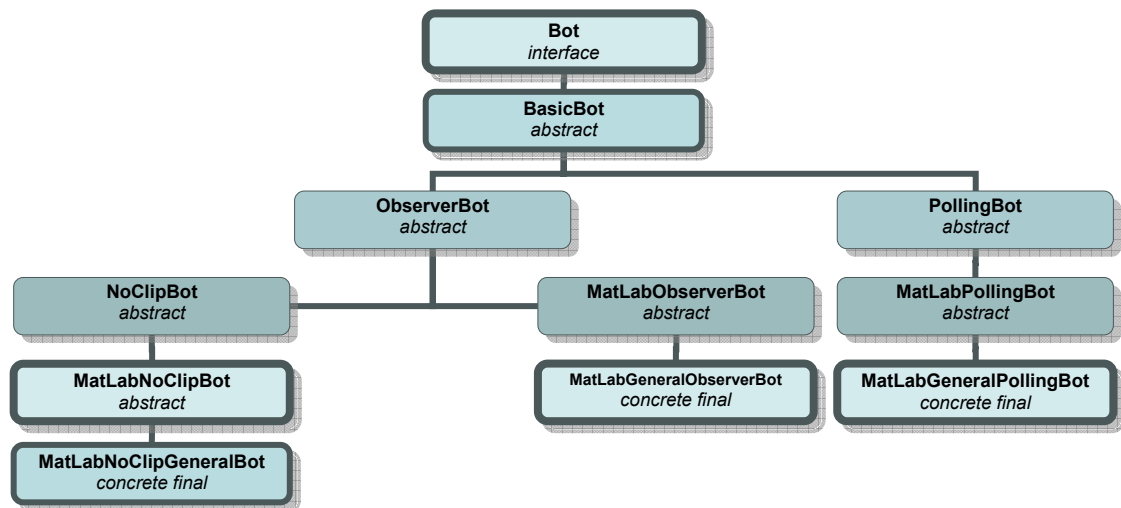


Figure 9 - The complete QASE Bot Hierarchy

5.1.1 Bot Interface

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

5.1.2 BasicBot

An abstract agent which provides most of the basic 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 write the AI routine in the predefined `runAI` method, and to supply a means of handling the server according to whatever paradigm he wishes to use; the third level of the Bot hierarchy provides ready-to-use implementations of two such paradigms. `BasicBot` also provides tailored, embedded access to the `BSPParser` and `WaypointMap` classes - methods in this class simply relay calls to the `BSPParser` or `Waypoint` object as appropriate, with certain parameters pre-defined to the most useful

values from the perspective of game agents (e.g. the bounding box used to trace through the level is set to the size of the agent's in-game character's bounding box). `BasicBot` will 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. Users can also obtain a pointer to the underlying objects, thereby allowing full access to their facilities.

5.1.3 *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 implements a method of detecting changes to the gamestate as indicated by their names, as well as a single point of insertion - the programmer need only supply the AI routine in the `runAI` method, while all other communication and data-manipulation requirements are handled by the API. Thus, the agent is notified of each update as it occurs, and a copy of the gamestate is presented to it. The user-defined AI routines then set the required movement, aiming and action values for the next update, and the API automatically 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. This means that the gamestate cannot be written in mid-AI cycle, but if the cycle takes longer than 100ms then subsequent updates from the server may be lost.
- 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` 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 ensures that no server frames will be lost, but may result in changes to the gamestate while the agent is executing its AI cycle. To prevent this, the bot can 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.

For full examples of how to write an agent, see the section “Creating QASE Agents”.

5.2 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 list of which items the player has collected and when they are due to *respawn*, automatically flagging the agent whenever such an event occurs.

Similarly, recordings of Quake 2 matches 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, 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.

5.3 DM2Parser and DM2Recorder

The API also facilitates the parsing of *demo* (DM2) files, which contain the full record of the human player’s activities during a particular match, via the *DM2Parser* class. A DM2 file is essentially an edited copy of the stream of network packets received during play, and therefore represents a complete encoding of the movements and actions of the player. DM2 files are organised into *blocks*, each of which consists of a header indicating the length of the block followed by a series of concatenated *messages*, as described earlier. The DM2 parser operates by treating the file as a *virtual server*, sequentially reading blocks and updating the gamestate as if it were receiving data online.

Furthermore, QASE incorporates a DM2 *recorder* which enables each agent to save 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. This is done by separating the header information (*ServerData* and *SpawnBaseline*) 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.

5.4 Environment Sensing

The network packets received 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 that of the various game entities. This information is contained in 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 currently visible), a means of locating, parsing and querying these map files is required. QASE’s *BSPParser* class provides these facilities.

Each Quake 2 map is constructed from a number of *brushes*, a generic term for any convex polyhedron such as a pyramid, cuboid, cylinder, etc; the term “convex” here means that any line through the object will have precisely one entry point and one exit point, the importance of which will be seen later. The planes formed by the faces of these objects divide the environment into a set of *convex regions*, or the areas of open space within (some of) which the actual gameplay takes place. Because Quake 2 levels are often very large and encompass a vast number of brushes, an efficient means of representing them and the convex regions they enclose is required; for this purpose, a *Binary Space Partition Tree* is used.

5.4.1 Binary Space Partitioning

The concept of a *Binary Space Partition Tree*, devised by Fuchs, Kedem, and Naylor [14], provides a parsimonious way of representing a virtual scene by specifying which other polygons lie *in front of* and *behind* the plane formed at the surface of a given polygon. The resulting tree structure can then be used to perform collision detection or to determine the order in which the polygons should be rendered, given the position of the viewer. A BSP tree

is created by first selecting a polygon to act as the *root node*, with an associated *splitting plane*; all remaining polygons are then classified as being in front of or behind this plane, and are added as its left and right children of the root node, respectively. In cases where a polygon intersects the plane, that polygon is divided in two at the point of intersection and both are added to the binary tree. The process is recursively applied to each of the child lists, gradually creating a series of subtrees until the list is exhausted. An example of BSP tree construction is shown below.

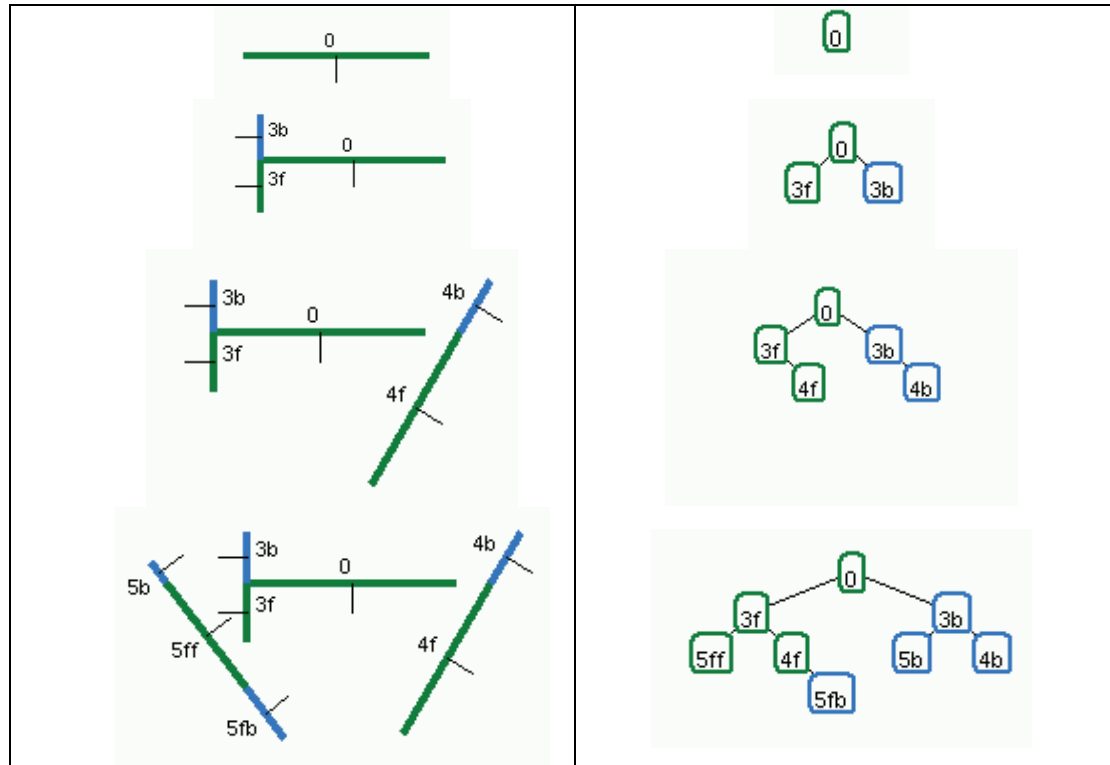


Figure 10 - Construction of a simple 2D BSP tree

5.4.2 Quake 2 BSP Format

Quake 2 BSP files consist of a *directory* indexing a series of *lumps*, each of which contains information about a different element of the environment (entities, textures, models, etc). From the perspective of obtaining sensory input about the game world, the most important of these are:

- the **Node** and **Leaf** lumps, each of which represents a single element in the BSP tree. Each **Node** contains an index to the plane equation used to split the tree at that point; this is stored as a *normal* with an associated *distance*. The **Leaf** object, representing a convex region within the level (including the spaces *within* objects), indexes each brush which bounds the empty space.
- the **LeafBrush** lump, which indexes the brushes (if any) surrounding each hull.
- the **Brush** lump, which contains information about each brush in the level. This includes the number of sides the brush has and its *content*, that is, whether it is a solid object (such as a wall) or may be passed through (water, mist).
- the **Brush Side** lump, indexing the *plane* associated with each side of the brush.
- the **Plane** lump, which provides the coefficients of each plane equation; these will later be used to determine the side of the associated face on which the player's avatar is currently located.

QASE's `BSPParser` class is used to load the contents of a Quake 2 BSP file, and to build the tree it represents. It also provides methods to perform *collision detection*, by sweeping a

4. Return back up the tree to the point at which the last branching decision was made, and continue with step 1 until all leaf nodes at which a collision may have occurred have been visited. The final result of the algorithm is the shortest distance between the start point and a collision with a solid subspace.

The above algorithm outlines the procedure for tracing a *line* through the game world; however, this may produce misleading feedback if the space through which the line passes is not large enough to accommodate the player, such as a window. To account for this, QASE also provides the ability to perform collision detection using a *bounding box* or *sphere* of predetermined or arbitrary size. The process is much the same, but collision detection is extended in each dimension to the outer extents of the volume. Thus, line-tracing can be used for visibility checks, while the bounding box approach determines the maximum distance which the player can travel in a given direction.

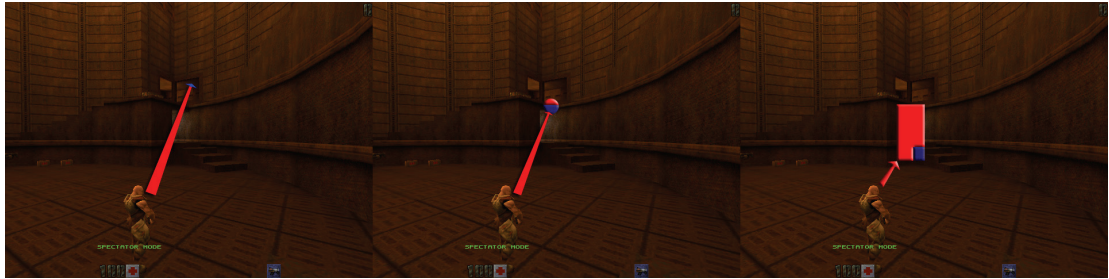


Figure 12 - BSP traces with line, sphere and box. Collision occurs at different points

Here, another advantage of using computer games in research becomes apparent - in many ways, they represent *idealised robot navigation worlds*. A great deal of work in this field is centred upon the elimination of *noise* in the robot's environment [15][17]; by contrast, computer games guarantee accurate, undistorted sensory information. Furthermore, map editors for such games are common, allowing the user to create any environment he desires. By utilising Quake 2 or other appropriate games as testbeds, researchers could easily prototype their navigation techniques without needing to account for such complications, before later adapting them to the additional constraints of the real world.

The environment-sensing functionality is embedded into the agents of the bot hierarchy above `BasicBot`; that is, they provide shortest-path methods which the agent transparently passes on to an underlying `BSPParser` object. For more information, please see the accompanying Javadoc.

5.5 MatLab Integration

The MatLab environment provides a rich set of vector- and matrix-handling routines, in addition to built-in toolboxes for neural computation, clustering, and other classification techniques; it also supports the use of the Java language. As such, it was an ideal candidate to provide an optional back-end engine 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 9** 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 quite 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 framework has proven very successful, both in terms of computational efficiency and ease of development; its advantage lies in the fact that custom bots can be written for each task, minimising the amount of work that both QASE and MatLab need to perform on each update. In contrast, the *MatLabGeneralBots* do not pass any information to MatLab or expect any results - the bot's entire AI cycle must be implemented from within MatLab.

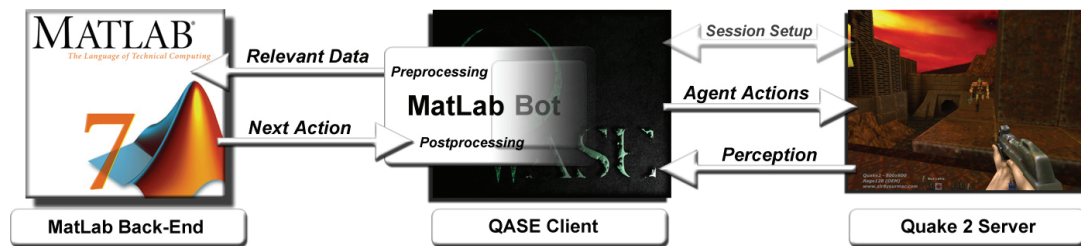


Figure 13 - *MatLab/QASE integration. MatLab acts as a back-end in the agent's AI cycle*

For a detailed tutorial on creating both standalone agents and those whose AI routines are implemented in MatLab, please see the chapter entitled "Creating QASE Agents".

5.6 Waypoint Map

The most basic requirement of any agent is an ability to negotiate its environment. While this can be done using the environment-sensing facilities outlined above, most traditional methods of navigation involve *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 topological 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 [4], this simply 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 [18] 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.

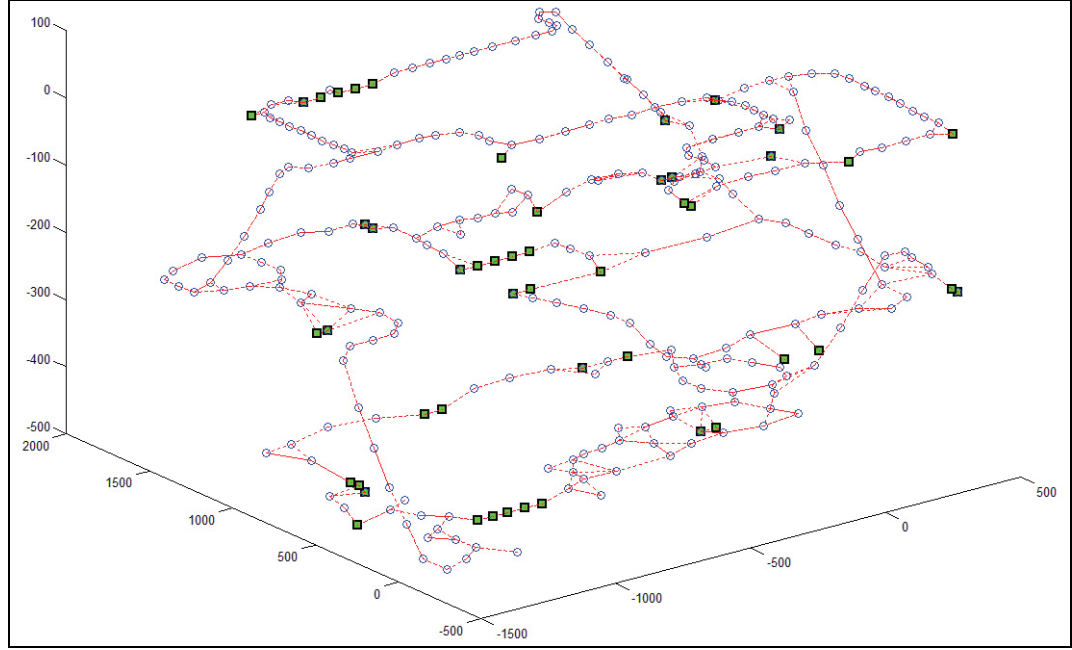


Figure 14 - A waypoint map, generated automatically by QASE from recorded human data and read into MatLab for visualisation. Green squares indicate item pickups.

The waypoint map functionality is embedded into the agents of the bot hierarchy above BasicBot; that is, they provide shortest-path methods which the agent transparently passes on to an underlying WaypointMap object. Additionally, the WaypointMap class provides methods allowing instances of itself to be saved to disk and reloaded; this enables users to generate a map once and use it in all subsequent sessions, rather than having to recreate it every time. For more information, please see the accompanying Javadoc.

5.7 Inbuilt AI Functionality

Aside from the waypoint map, QASE also incorporates a number of other AI structures. These features are included primarily for education purposes, to allow students to examine and experiment with AI constructs commonly found in undergraduate curricula - for more demanding research applications, the MatLab back-end can be used. More information on each of the following classes can be found in the accompanying Javadoc documentation.

5.7.1 K-Means Clustering

The *k-means* algorithm is an iterative procedure used to estimate the best reference vectors given a sample dataset - that is, those which minimise the reconstruction error. A k-means calculator package (`soc.qase.ai.kmeans`) is included, to serve as an illustration of the concepts and implementation of clustering techniques; it is also used extensively in QASE's *waypoint map generator*, to derive the waypoints from raw observation data. The k-means facilities are composed of two classes, `KMeansCalc` and `KMeansData`. The latter acts as a wrapper for the data output by the algorithm, whereas the former performs the actual clustering according to the following formulae:

$$b_i^s = 1 \text{ if } \|x^s - c_i\| = \min_j \|x^s - c_j\|, \text{ and } 0 \text{ otherwise}$$

$$c_i = \frac{\sum_s b_i^s x^s}{\sum_s b_i^s} \text{ while } c_{i_{t-1}} \neq c_{i_t}, i = 1 \dots k$$

where k is the number of clusters, x is a point in the dataset, and c is a cluster centroid.

5.7.2 Genetic Algorithms & Neural Nets

For education purposes, QASE incorporates implementations of both a *neural network* and a *genetic algorithm generator*, found in the `soc.qase.ai.gann` package. These are designed to be used in tandem - that is, the genetic algorithms are applied to gradually evolve the neural network's weights according to 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 at hand. The class also allows the user to save an entire simulation to disk, and resume it from the same point later.

6. Creating QASE Agents

In this chapter, 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.

6.1 Standalone QASE 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 available item in the game environment, collect it, and move on.

```
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); // set movement dir
    }
}
```

Figure 15 - Extract from *SampleObserverBot.java* (some comments added)

6.2 QASE MatLab Agents

QASE agents which use MatLab as a back-end engine fall into one of two subcategories, as mentioned in the “MatLab Integration” section earlier - 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 with the API.

6.2.1 Importing QASE

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 16 - The `prepQASE.m` script. Imports common packages into the MatLab environment

6.2.2 Direct ML 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 `MatLabGeneralPollingBot`, as the former gives superior performance; no such performance discrepancy exists in the case of hybrid agents.

```
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 17 - 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.

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. The code of this script is given below:

```
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
    end
```

Figure 18 - *SampleBotManagerGeneral.m*. This closely parallels the *SampleObserverBot.java* source shown in the “Standalone QASE Agents” section 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*).

6.2.3 Hybrid 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), thereby 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 at hand 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 *preprocessing* 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 script 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.

The code of the `BotManagerTemplate.m` script is given on the following page.

```

function [] = BotManagerTemplate(botTime, recordFile) % botTime specifies the
bot's lifetime in seconds (-1 for infinite)

    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
end

```

Figure 19 - The *BotManagerTemplate.m* script. Users need only define the size of the *mlResults* cell array (ie the number of outputs) and supply the app-dependent computations as indicated.

Again, a sample hybrid agent is provided; the relevant MatLab function, *SampleBotManager.m*, is show below. As can be seen, hybrid agents generally result in far tidier and comprehensible scripts. The most important aspect of this example is that MatLab both *receives* and *returns* native vectors - it performs no Java manipulation at all.

```

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 20 - The *SampleBotManager.m* 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, again in the form of a MatLab float vector (which is auto-converted into a Java float[] array).







References


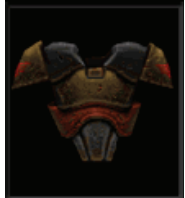


- [1] Weiss, G., "Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence", The MIT Press, 1999.
- [2] <http://www.idsoftware.com>
- [3] <http://www.javasoft.com>
- [4] Gorman, B. and Humphrys, M. "Towards Integrated Imitation of Strategic Planning and Motion Modelling in Interactive Computer Games", in Proc. 3rd ACM Annual International Conference in Computer Game Design and Technology (GDTW 05), Liverpool, Nov 2005, pages 92-99
- [5] Gorman, B., Fredriksson, M. and Humphrys, M. "QASE - An Integrated API for Imitation and General AI Research in Commercial Computer Games", in Proc. IEEE 7th International Conference on Computer Games: AI, Animation, Mobile, Educational & Serious Games, Angoulême, November 2005, p 207-214
- [6] A. Naraeyek. Computer Games - Boon or Bane for AI Research. Künstliche Intelligenz, pages 43-44, February 2004
- [7] Tesauro, G. (1994). TD-Gammon, a self-teaching backgammon program achieves master-level play. Neural Computation, 6, 215-219
- [8] Bauckhage, C, C. Thureau & G. Sagerer (2003): Learning Humanlike Opponent Behaviour for Interactive Computer Games, in Pattern Recognition, Vol 2781 of LNCS Springer-Verlag
- [9] Laird, J. E. and v. Lent, M. (2000). Interactive Computer Games: Human-Level AI's Killer Application. In Proc. AAAI, pages 1171-1178
- [10] J. E. Laird. Using a Computer Game to develop advanced AI. IEEE Computer, pages 70 -75, July 2001.
- [11] J.E. Laird, It Knows What You're Going To Do: Adding Anticipation to a Quakebot (2000), Proceedings of the Fifth International Conference on Autonomous Agents
- [12] JE Laird and John C. Duchy, "Creating Human-like Synthetic Characters with Multiple Skill Levels: A Case Study using the SOAR Quakebot", AAAI 2000 Fall Symposium: Simulating Human Agents, North Falmouth, MA
- [13] Sklar, E., AD Blair, P Funes & J. Pollack, 1999: Training Intelligent Agents Using Human Internet Data, in Proc. 1st Asia-Pacific Conference on Intelligent Agent Technology
- [14] Fuchs, H., Kedem, Z., and Naylor, B., On Visible Surface Generation by A Priori Tree Structures, Conf. Proc. of SIGGRAPH '80, 14(3), 124--133, 1980
- [15] J. Borenstein and Y. Koren. Real-time obstacle avoidance for fast mobile robots. IEEE Transactions on Systems, Man, and Cybernetics, 19(5):1179--1187, - 1989
- [16] J.M.P. van Vaveren. The Quake III Arena Bot. Master's thesis, TU Delft, June 2001.
- [17] Koren, Y. and Borenstein, J., 1991, "Potential Field Methods and Their Inherent Limitations for Mobile Robot Navigation." Proceedings of the IEEE International Conference on Robotics and Automation, Sacramento, California, April 7-12, 1991, pp.1398-1404
- [18] Floyd, R.W., Algorithm 97, Shortest path, Comm. ACM. 5(6), 1962, 345


Appendix: Fixed Entities



As described in the previous chapters of the specification, a simulated environment used by the QASE application programming interface consists of moving entities (agents) and fixed respawning entities. Below you will find an outline of the fixed entity types: weapons, armor, health, ammunition, and special items. Also, each category has a number of unique attributes (e.g. weapon range, armor shielding, healing, ammunition volume, item duration, etc.) associated with it.








Weapon	Description	Uses	Range	Damage	Respawn
	Blaster	None	Close	15 / shot 2 shots / second	30 seconds
	Shotgun	Shells	Medium - Long	4 / shell 12 shells / shot 1 shot / second	30 seconds
	Super Shotgun	Shells	Short	6 / shell 20 shells / shot 1 shot / second	30 seconds
	Machine Gun	Bullets	Medium - Long	8 / bullet 10 shots / second	30 seconds
	Chain Gun	Bullets	Short - Medium	6 / bullet 20 - 40 shots / second	30 seconds



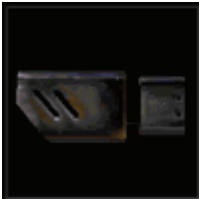
Weapon	Description	Uses	Range	Damage	Respawn
	Hand Grenade	None	Short	125 / grenade 1 / 2 seconds	30 seconds
	Grenade Launcher	Grenades	Medium	100 / grenade 1 grenade / second	30 seconds
	Rocket Launcher	Rockets	Medium - Long	100 - 120 / rocket 5 rockets / 4 seconds	30 seconds
	Hyper Blaster	Cells	Short - Medium	15 / cell 10 cells / second	30 seconds
	Rail Gun	Slugs	Long	100 / slug 1 slug / 1.5 seconds	30 seconds
	BFG10K	Cells	Short - Medium	0 - 20 / cell 50 cells / shot 1 / 2 seconds	30 seconds

Armor	Description	Shielding	Respawn
	Jacket	25	20 seconds
	Combat	50	20 seconds
	Body	100	20 seconds
	Shard	1	20 seconds

Health	Description	Healing	Respawn
	First Aid	10	30 seconds
	Medkit	25	30 seconds
	Stimpack	2	30 seconds

Ammunition	Description	Associated Weapon	Volume	Respawn
	Shells	Shotgun / Super shotgun	10	30 seconds
	Cells	Hyper blaster / BFG10K	50	30 seconds
	Bullets	Machine gun / Chaingun	50	30 seconds
	Grenades	Grenade launcher	5	30 seconds
	Rockets	Rocket launcher	5	30 seconds
	Slugs	Railgun	10	30 seconds

Special Item	Description	Duration	Respawn
	Energy Armor. This provides improved protection against energy weapons. While it is being used, it drains energy from your cells when damaged.	∞ (if cells are available)	60 seconds
	Bandoleer. Increases carrying capacity of shells (100 to 150), bullets (200 to 250), cells (200 to 250), and slugs (50 to 75).	∞	
	Heavy Pack. This allows you to carry more ammo on your back.	∞	60 seconds
	Underwater Breather. This provides oxygen when submerged in liquids.	30 seconds	60 seconds
	Environmental Suit. This protects you against damage from hazardous liquids, such as Slime.	30 seconds	60 seconds
	Quad Damage. The quad temporarily multiplies all your weapon's strengths by four times.	30 seconds	60 seconds
	Mega Health. The mega health provides a temporary but significant boost to your health.	60 seconds	Depletion + 20 seconds

Special Item	Description	Duration	Respawn
	Invulnerability. The invulnerability item renders you temporarily indestructible.	30 seconds	300 seconds
	Super Adrenaline. This slightly increases your health permanently.	∞	60 seconds
	Silencer. This silences the discharge of any weapon.	Silences 30 shots for each gun, except the chain gun (60 bullets) and the BFG10K (15)	60 seconds