Quake II as a Robotic and Multi-Agent Platform

Chris Brown, Peter Barnum, Dave Costello, George Ferguson, Bo Hu, Mike Van Wie

The University of Rochester
Computer Science Department
Rochester, New York 14627

Technical Report 853

November 2004

Abstract

We have modified the public-domain Quake II game to support research and teaching. Our research is in multi-agent control and supporting human-computer interfaces. Teaching applications have so far been in an undergraduate Artificial Intelligence class and include natural language understanding, machine learning, computer vision, and production system control. The motivation for this report is mainly to document our system development and interface. Only early results are in, but they appear promising. Our source code and user-level documentation is available on the web. The information document is a somewhat motion-blurred snapshot of the situation in September 2004.

Keywords: autonomous agents, multi-agent coordination, multi-agent collaboration, robot simulation, Quake, implemented systems, teaching AI, open source

This work funded by the UR Computer Science Department, NSF Research Infrastructure award for "Spatial Intelligence" EIA-0080124, SRI International Subcontract 03-000223, and NSF award IIS-0328810.

Contents

1 Animats, Research, and Teaching

U. Rochester's Computer Science Department has active research activities in mobile robotics (e.g. [32; 59; 45; 46]), computer vision (e.g. [60; 61; 56; 32; 53; 40; 51; 54]), multi-agent cooperation (e.g. [75; 76; 74; 72; 73; 23]), natural language and dialog understanding (e.g. [15; 24; 19]), command interfaces and human-computer interfaces for mixed-initiative planning and multi-agent systems (e.g. [17; 18]). Common to all these topics is the notion of intelligent agents: entities that act autonomously in response to changes in their state or the state of their environment. Such agents are becoming ever more central not only to our research but to our teaching. After this short introduction we go into some detail about our Quagent system and how it is being used in both endeavors.

Our quagents are a form of *animat:* that is, a simulated organism [69]. Among the most famous are Tu's artificial fish [70]. A disadvantage to animats is that no one can really simulate natural complexities in a scientifically respectable way. If we want AI systems to transfer simulation-gained capabilities into the real world, then an adequate simulator can be as much trouble to create and deal with as real hardware. The main animat advantages are: no hardware, known ground truth, ability to create wide variety of (physically possible or not) sensors and environments, no hardware, quick prototyping, and no hardware (amongst other obvious ones.) Further, most AI techniques never worked in the real world and never will – so what?

Laird and co-workers have used the Soar production system to simulate various human reasoning characteristics and skills in a "game AI" [49; 47]. That is, the Quake "monsters" (automated opponents of the first-person player) had their inbuilt "AI" replaced with the same goals in mind: competition with human opponents. The Soar Quakebots have human-like limitations – for instance they build maps through exploration. In a recent version, the Soar Quakebot was played with independent variations in the four dimensions of decision time, aiming skill, aggressiveness, and tactical knowledge. This bot has 715 rules and 100 operators [48]. Judges rated the "humanness" of the various versions of the automated players.

Quake III has been used as a testbed for neural models [10]. Besides their influence in mainline AI research (also see [8]), games are making inroads into all sorts of relevant related areas. Machinima is a fascinating new art form (e.g.[9]). Systems administration makes for a possibly lively application (e.g. [4]).

We have been using animats in various ways. For research, Boston Dynamics' DI-Guy system is installed in our Virtual Reality laboratory. It has provided a good platform for research in psychophysics, and more to the point in animat vision both here and elsewhere [65; 68; 64].

Also, we wanted a (mostly) open-source system that ran on a variety of platforms. We believe the Quake-II system is comparable to the DI-Guy system in commandability. Typical DI-Guys look like humans. It is not easy in either system to get at the kinematics (much less dynamics) of the agents; it is much easier to let them move and appear as they do in their pre-programmed modes. The DI-Guys system is mostly graphics, and does not have the inbuilt level of physical simulation we wanted.

In **research**, we want to use Quagents to study

- Strategies for flexible, self-organizing cooperation possibly heterogeneous intelligent agents, especially with minimal communication.

- Methods of commanding and communicating with such teams.

In general, we are most interested in distributed coordination of intelligent agent teams (in general heterogeneous agents with different physical and computational abilities) using constrained (in the limit, only passive visual) communication. We assume that re-planning and re-assignment of roles can and should happen based on higher-level re-tasking or local intelligence or events, that agents can defect or fail, that communication should be minimized, and that visual information can sometimes substitute (this raises the nontrivial (well, unsolved) "vision problem", however).

We would like to allow individual agents to make their own decisions regarding how best to carry out their tasks, and to have the formalism apply to teams composed of a wide variety of autonomous, unmanned ground, air, and sea vehicles, possibly with very different physical and computational abilities.

In **teaching**, we have strong graduate and undergraduate AI teaching programs with the usual array of regular courses, topical seminars, and courses manufactured around large-scale efforts like the undergraduate robot team [59; 45; 46]). Our mainline undergraduate artificial intelligence course uses Russell and Norvig's text *Artificial Intelligence – A Modern Approach*, whose tagline is "The Intelligent Agent Book" [58]. We have used the RoboCup simulator in a special session of the AI course for an extra hour of credit but that was a one-time experiment.

For teaching it seemed reasonable to take advantage of the fact that many undergraduate majors like games, that a fair amount of work is going on to improve AI in games, and that there is increasing awareness that standard AI topics can be coupled to games [39; 31]. (In fact game-programming is the domain or a recently-initiated software engineering course here, which has support from the Kauffman Institute under an entrepreneurship grant to the U. of Rochester... but that is another story).

In the past we have used student-written gridworld robot simulations, often created in Matlab. We created the *Quagent* (Quake Agent) system described here to meet the demands of both research and teaching. We wanted to get something like real physics and real 3-D rendering in an (arguably) exciting domain without paying the price of writing our own simulator.

In what follows we discuss two pieces of research in controlling agent teams, describe our teaching applications, and supply an increasingly deep technical look at the Quagent system and how to use it. Appendices deal with further software details, and our Quagent website [14] by definition has the most up-to-date material, including downloadable code.

2 Planning and Role Selection by Observation

Van Wie's Rhino system (named for Rochester's soccer team) is not yet instantiated in Quagents, having been conceived and implemented in the simulated RoboCup (soccer) context [72; 73]. At the time the robocup simulator had the advantages that Quagents offer now: stability, documentation, physics, instrumentability. Van Wie's work was among the first to tackle the formal problems of such passive self-organization, but the field has taken off (e.g. with conference workshops [71]). It is concerned with the assignment to individuals of roles in a group plan, after which role trading may take place. Rhino assumes no symbolic communication among agents at all. It substitutes observation for communication. It is based on firm probabilistic formal foundations, and can be used to monitor the progress of a group (and, if necessary, re-assign roles) as well as to assign the initial roles.

Rhino team members have in common a repertoire of pre-existing plans, or a “playbook”. There are obvious sports analogies, and military analogs involve rules of engagement and doctrine for how to run a sniper team, surveillance patrol, building search, etc. Given these prior role definitions, the teams will not have to re-invent well-known strategies and tactics: Rhino does not learn plays, though it has a learning component for learning activities.

Rhino provides an efficient combination scheme that maps single-agent plan hypotheses into group plan hypotheses, and it uses a cost-benefit action selector to avoid role conflicts and resolve synchronization problems. Rhino has real-time detection and repair of teamwork failures (defections and action failures), which also need a belief revision methodology to revise beliefs on failure. The plan-recognition algorithm is probabilistic, robust to sensor errors, and allows both top-down and bottom-up reasoning directions for efficient guidance and opportunistic discovery.

Without symbolic communication, distributed plan formation and role assignment requires recognizing teammates’ (and opponents’) actions and thus inferring their probable plans. This capability involves other hot topics today, group and individual activity recognition, and person recognition. temporal texture, object discovery and change detection. The recognition algorithms are trained from real-world examples.

Van Wie’s formal results and algorithms are representative of local research that should prove useful and relevant to future efforts realized in the Quagent system.

3 Group Planning and Acting with Negotiation

Barnum’s recent work with the Quagent simulator implemented role-assignment, role-swapping and resource-allocation methods that approach optimal with good communications and degrade gracefully as communications degrade. Each robot is made responsible for a subset of the team’s work, but while pursuing their goals, the robots attempt to trade goals with others that are more capable, thereby improving team performance. As communications degrade, however, trading becomes less frequent, and in the worst case (that is, no communication at all) robots are still able to perform their own tasks without trading them. Coordination is truly distributed, not hierarchical. Each is able to perform its tasks without communication, only trading with others if a link is available. The overarching control system is independent of individual robot design, so is applicable to teams composed of a wide variety of autonomous, unmanned ground, air, and sea vehicles. In the limit of this formalism robots have neither communication with nor observations of other robots, cooperation ceases, but assigned tasks are carried out (however inefficiently). This work involved some extensions to the original Quake Protocol Barnum inherited, and some (now optional) changes to the Quake engine to allow communication between agents (Section 9).

Tolerance of disruption and the ability dynamically alter tasks dynamically is vital for any successful multi-agent team. It is possible to have a very robust system by not allowing agents to communicate, but complex tasks and efficient coordination require it. When groups of robots are interested in common goals, they must be able to work together, or else it may take a long time to complete their tasks. But at the same time, they should have enough autonomy from each other that they do not have to rely on communication when it cannot be guaranteed.

Many team coordination systems make the assumption that each unit of the team has a fixed set of capabilities. We are influenced by the idea of software agents that migrate between computer systems (web crawlers are an example). By using large groups of software agents and small groups of hosts, we have been able to design a robust and dynamic teamwork system. Any implementation with multiple software agents can take advantage of the intrinsic failure tolerance and dynamic goals such a design brings. The designer only has to decide which single-goal agents are needed for the system to succeed and a way to calculate their utility. Human interaction is simple too; people can interact with the system in a goal based way, either adding or removing agents (and therefore goals,) and leaving the optimization to the system.

3.1 Hardware Hosts and Software Agents

In this section, our nomenclature (indeed, taxonomy?) becomes confus(ing or ed). Agent-hood depends on the domain: a software agent (softbot, even robot) works in the electronic domain of programs, a hardware agent works in the world. In this section, we envision programs to solve tasks, carry out roles, run behaviours, etc., as active things that can migrate where they are needed [36; 38]. In that sense they are software agents. Once they are in place, they form the control for hardware bots (they become part of the quagent controller). However the quagent in this view can be considered simply the execution shell for the more active agents, and hence is referred to here as a *host*. Thus our quagents, which control hardware, become *hosts* to software *agents*. The latter are like the plays in Van Wie's agents' playbooks [73]; they are behaviours, skills, tools, roles, available to the host.

Just as Van Wie's agents do a cost-benefit analysis to decide which play from the playbook to assume, in this work a host conducts an auction in which each resident agent submits a bid of its estimated utility.

A *mobile (software) agent* is simply a software agent that has the ability to move with its state intact to a different machine, which allows efficient resource exploitation [55]. Mobile agents are most useful when communication is either limited or uncertain since they can get "close to the action" to minimize communication. Any system where the code is separated over long distances or is plagued with malfunctioning networking can benefit from locally executing code. Mobile agents provide a generalized wrapper that can operate on various hardware platforms, without requiring the designer to explicitly write code for them.

3.2 Communications

There are a variety of systems that use auctions, which can be fast and unambiguous, to coordinate the behaviors of multiple agents [78; 77; 62; 35; 37] We are using an auction system for the coordination of several agents within a single host, with the host acting as the auctioneer, taking bids on the rights to use its resources. To participate in an auction, an agent calculates how much the auctioned item will benefit its utility, and sends a bid to the auctioneer. The auctioneer is not required to know why any of the bids are what they are, only that it should select the highest. All of the understanding of desires and abilities can be communicated by a single number.

Although the communication costs within a single machine are irrelevant, the need for the coordination of different tasks is still present. By encapsulating the calculation of utility within each agent, the host only has to choose the one that reports having the highest numerical value. This way, the host does not have to have any knowledge of either the data or thought process of the utility calculation.

The usable set of agents can constitute a host's static playbook, or agents can move between hosts. Agents can rank hosts as to their suitability for the agent's purposes and migrate to the best host. The decision to move uses the same measure of utility as the auctions, an auction is not involved and it can travel to any host it wishes. Each agent should attempt to move to the host that will give it the highest utility, using information on the structure and state of the host, as well as how busy the host is with others.

The communication protocol is similar to the contract net [62; 34], with two main differences. One is that once an agent (and therefore goal) is transfered to another host, the first host does not have to monitor its progress to make sure it is completed, reducing communication overhead. The second is that for a task change, contract nets use three sets of communications, while our system uses two. The downside of our system is that each host has to update the others every time it completes a state change, more in the manner of a specific sharing system [36]. As a result, the total amount of communications is not substantially less, is more spread out and does not tend to be bursty. Broadcast messages to all agents would lower the number of messages substantially, but even with point to point communication, our method can utilize communication hardware efficiently.

A reliable agent-transfer protocol deals with lost messages (details in [23].) As a result, agents may be replicated but not lost. The movement of the agents from one host to another is greedy, so performance is not guaranteed optimal. Our experience confirms the sentiment that first-choice strategies such as auctions tend to work reasonably well [34].

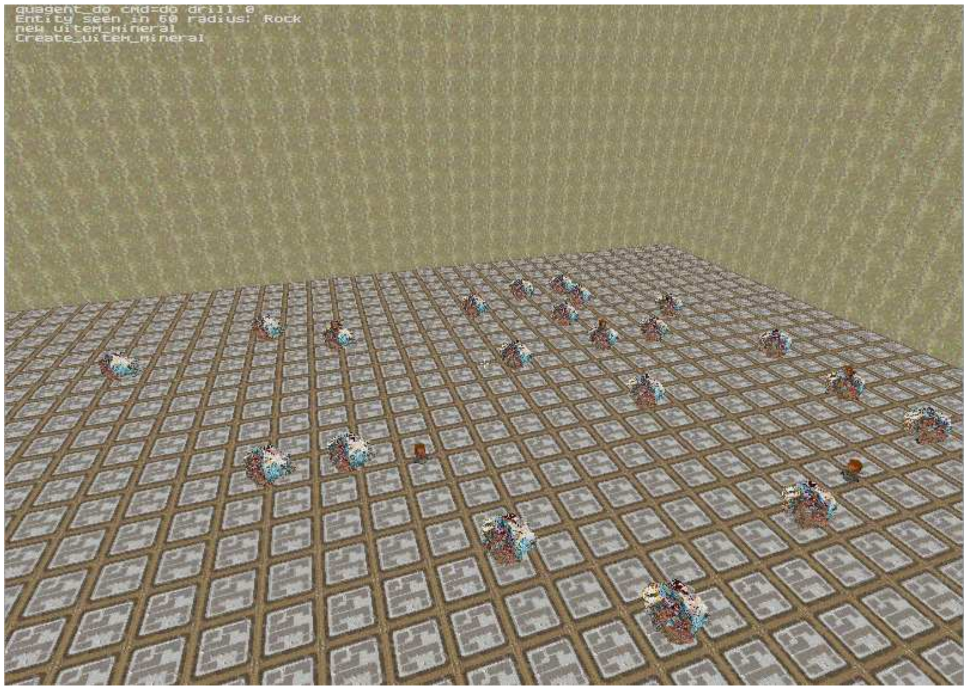
3.3 A First Demonstration

To build the necessary infrastructure and to test our proposed role-allocation method, We (i.e. Barnum) created a simulated world with the core decision making and agent transfer components. In the simulation, there is a large group of randomly positioned rocks that have to be drilled for samples (Fig. 1). We used from one to five simulated homogeneous mechanical rovers, and assumed a different individual software agent responsible for the drilling of each rock. Each of the rovers is a host, and has a list of all its resident agents, and simulated communication hardware to communicate to other hosts. The fitness of a host to drill a particular rock simply depends on its being untasked and on the distance to the rock.

This scenario does address the efficiency of multiple coordinated parallel individual actions, but does not address the issue of multi-actor coordination. That is, no task requires more than one actor to accomplish, and *a fortiori* no task requires multiple actors playing different roles.

Our mobile agents are hybrid automata (automata controlled by both discrete and continuous state values) that control a rover to perform a goal. Each automaton directs an host robot to move to a rock, then to drill it, with tolerance to somewhat uncertain motor control and collision avoidance. (For more details on the design and function of the automata, see [23]) The automata are designed so that if a rock turns out to be non-drillable, already drilled, or unreachable, then that agent will remove itself from its host.

In most of our experiments each host has a playbook of all agents and agent code is not copied in role-swapping, rather agents are uniquely identified by their goal. This is a shortcut that is equivalent



```
quagent_do cmd=do drill 0
Entity seen in 60 radius: Rock
new uitem_mineral
Create_uitem_mineral
```

Figure 1: Agents in action. Operating surfaces are textured to help humans gauge distance. Rocks are the larger objects, host drilling bots appear smaller and dark.

to free and reliable communication for sending programs around the system. Agents also need to be able to share data, mostly related to their host's positions.

While one agent has control of a host, the host's other agents are free to move around to other hosts. To gauge whether and where to move, agents need to know the locations of the hosts. Hosts are rarely idle, and so the agent currently in charge of a host has to give all other hosts an estimate of where it will end up and how long it will take to get there (details in [23]).

3.4 Analysis

We ran instrumented trials to test our implementation (details in [23]). We varied the number of agents in the team, tested different levels of communication uncertainty, and used several randomly generated groups of rocks, etc. We quantified team effectiveness, message traffic, (e.g. Fig. 2). On average, as more agents are added and as communications are more reliable, the faster the problem gets solved.

3.5 Future Work

Barnum's work is also in support of research on command interfaces to intelligent teams: he is working with the Conversational Interaction and Spoken Dialog Research Group at URCS, whose goals are the development of mixed-initiative planning algorithms and intelligent software agents interfaced by natural language to a human partner [3]. The current interface (e.g. to the TRIPS,

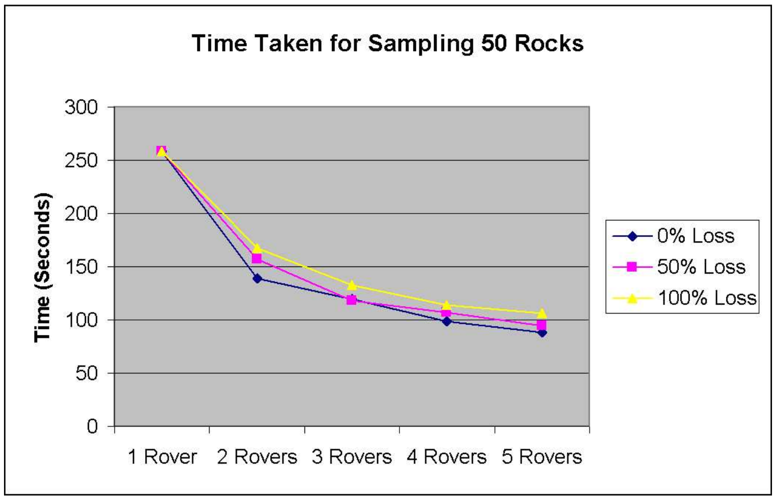


Figure 2: Time taken to complete

TRAINS, and MedAdvisor dialog-understanding systems) uses speech, graphics, menus, and some work has been done on monitoring eye movements. The Quagent domain interface raises new issues because of the multiple agents involved and the nature of the tasks. Our aim is to implement commands by by goal-setting at the highest level possible, leaving lower-level organization to the swarm. One basic subtask is to map the English commands to the playbook entries.

The algorithms and techniques we create will be useful under a wide range of assumptions about agent capabilities. The visual capabilities can be trained by examples. The agent's playbooks can be developed and tested offline (in simulation, say). We picture a toolbox for configuring multi-agent missions in many domains.

We must expand the problem domain to include individual subtasks requiring multiple (multi-role) heterogeneous actors. In the drilling domain, John Henry could need a shaker, (multiple roles) or it may take two or three robots in similar roles to carry an object. GRASPlab has been a leader in coordinating real robots using hybrid control systems (e.g. [63; 13].)

Perhaps one of the most promising future avenues is to refine the structure of the agents themselves. Although hybrid automata allow for complex function, hierarchical decision making, as in [29; 67], could allow for much more robust single-agent functionality even within the current teamwork system. Another more difficult addition would be to adapt the auctioning and the decisions for an agent to travel from one host to another. Combinatorial auctions have been successful in many agent-based and planning approaches [20; 42; 50], and give better performance than simple greedy searches.

This work can be melded with Van Wie's (and others') approach, which suggests that performance can be guaranteed for robots by having them observe their environment and see what their teammates are doing instead of explicitly communicating [72; 73; 41; 22; 71].

To further reduce the need to rely on communication, elements from swarm robotics, mechanism design, and stigmergic theory could be added or substituted [57; 66; 30]. Even though the agents themselves move from host to host, it may be possible to make the interactions simpler, so as to allow for greater functionality, even in uncertain environments.

Whatever communication scheme is attempted, it would be beneficial to use a more evolved agent communication language, such as [5]. Although our current coded system is functional and efficient, a different method could allow for better coordination of teamwork, and possibly even the ability to have agents work directly together on joint plans, using ideas such as joint intentions [43].

Any of these methods could allow for increase in performance and usability. There are as many ways to improve the design as there are to design it in the first place, but most likely those that warrant the most study are with *emergent behavior* and *hierarchies*. Both allow for agents that can perform in uncertain and hostile situations without severe penalties if the environment is other than expected, or agent failures occur. The ultimate goal of an autonomous team that can handle the unexpected can only be possible by a merging of techniques.

Last, UR's Computer Science Dept (URCS) has a few mobile robots on which algorithms involving a small number of robots could be demonstrated [59; 45; 46].

4 Pedagogical Quagents

4.1 Background

At UR we are lucky to have an unusually strong AI component in our history and in our faculty. We have a record of achievement in robotics and vision, and also in natural language understanding, logic, and planning. We also have a longstanding cordial and productive relationship with our faculty in parallel and distributed systems. In 2004 we received the UR's Goergen award for undergraduate curricular development, largely based on our record of involving undergraduates in research in a significant way.

The Quagent system is designed to support teaching as well as research. In keeping with the tagline ("The Intelligent Agent Book") of our text [58], with the predictable interest of our undergraduate students in computer games, with the recent introduction of a software engineering course based on such interactive games, and with our own experience with physical mobile robot courses at the graduate and undergraduate level, our goal is to have quagent versions of all our programming exercises for one of our undergraduate AI courses.

The following section summarizes the detailed assignments referenced in the main course assignment web page [1]. They follow the order of presentation of topics in the text and are sometimes based on text exercises. The course follows the order of the text chapters reluctantly treating some more lightly than others [2]. The projects fall naturally into the order they are introduced in the text, which moves from symbolic state-space formulations through logic and probabilistic representations to climax with several good chapters on various forms of computer learning. Unlike many AI texts, Russell and Norvig do a fine job with robotics and vision as well. Thus our programming assignments cover state-space search, production system robot control (*e.g.* subsumption-type controllers), probabilistic techniques and computer learning algorithms, natural language understanding and generation, and several options in computer vision.

4.2 Project Summaries

At this point we invoke a

Standard Disclaimer: The best definition of Quagent behavior is the code. Next best is the web documentation (all pedagogical Quagent web pages spring from the main Quagent page [14].) Sketchiest, most approximate and abstract is the treatment in this TR.

All student project assignments (and some sample projects from past years) spring from [1]. Projects are normally to be undertaken in teams of two. These project descriptions rely mildly on an understanding of Quagents and the Quagent protocol (Section 6).

State-Space Problem Solving

Write general state-space search algorithms (depth first, breadth first, A*. etc.) and use them to solve a Quagent maze-running application. In previous years we have used Jug-filling problems as well as other search problems culled from Sam Loyd's puzzles [52]. In the past the search routines had to be applied to than one application domain, so it would be good to find another problem besides the maze for the Quagents. The maze problem will necessitate creating some specialized "levels", or Quake environments, but that should not present any major difficulties (Section 6.7).

Production Systems

This project is to learn about production systems and to experience interacting with current commercial-grade AI software. We use Jess [7], and our code distribution includes a Jess example (See Appendix A). The exercise also introduces the possibility of quagent-quagent communication.

Fig. 3 shows a map produced by a Quagent under production system control.

Students write and demonstrate one or more production system controllers (using Jess and probably Java) for one or more Quagents. The Quagent Protocol (Section 7) is used for sensing situations and issuing commands. Controllers are to be compared quantitatively and qualitatively. Likewise, the environment can be changed The bots interact with items in different ways, and students can change the experiment by using the configuration files to change situations. For instance, by adding more obstacles (BOXes).

Students do not have to use all the bot's facilities. The STOPPED message is like a bump sensor and it's interesting to write rules to continue moving in a straight line despite bumping into and maneuvering around obstacles using only such a sensor.

In pre-Quagent years, students usually implemented a subsumption architecture (e.g [25; 27; 28; 26], with at least four behaviours.

Obvious sorts of primitive behaviors that come to mind are the following.

- Sense and chase another bot or client (sex, curiosity, or general playfulness).
- Avoid obstacles while trying to walk in a straight line.
- Change direction when prematurely stopped, or generally try to fix the situation.
- Follow a wall (could use last behavior maybe?).
- When energy goes low, find a battery and pick it up.

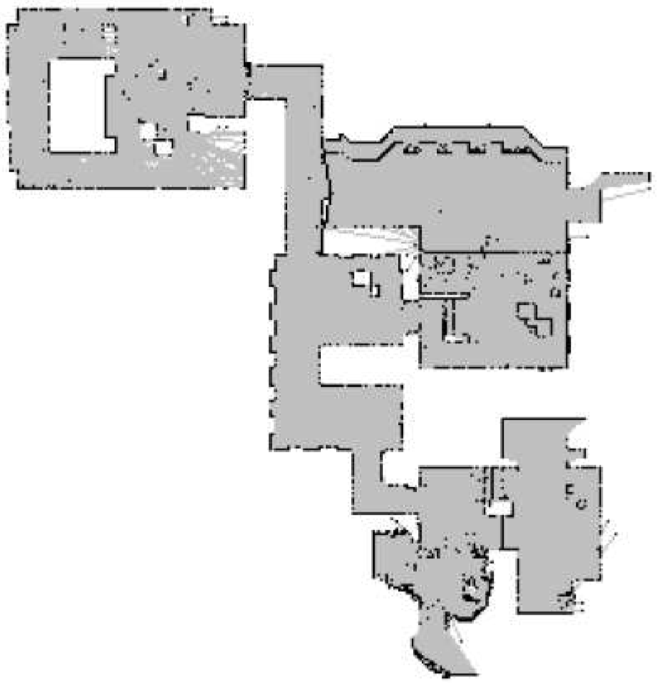


Figure 3: Map made under production system control.

- Avoid a static or randomly-moving ”predator” (other bot or the client (Fear).
- Accumulate wisdom to improve visual resolution (with RAYS).
- Maintain and monitor other resources like wealth or health and try to maximize it and do the right thing when it is depleted. Presumably find the right resource and pick it up.
- Explore randomly (Curiosity).
- Collect attractive items (Greed).
- Avoid kryptonite.

The controller can be responsible for some interesting quagent-quagent interactions. For instance it can simulate a version of trading or selling. The controller would know that bot1 has an excess of a commodity (like HEADs) and not a lot of GOLD. If there were another bot with more gold than heads, then controller could arrange for them to meet and exchange goods with some pickups and drops.

Quagents can thus communicate directly with each other (through their controllers) even there is no mechanism for the bots to talk to each other. The quagent controllers can effectively implement broadcast or point-to-point communication between bots. Quagent controllers can engage in complex activities like negotiation, bluffing, whatever.

Learning

Here students proceed from trying different learning algorithms on a small abstract 3×4 gridworld to using such offline learning to create policies for “real world” quagents, to doing on-line learning in the Quagent simulator.

The first part is much like exercises 17.6 and 21.7 of the Russell and Norvig text [58] Ex. 17.6 uses the 12-cell Wumpus World to experiment with policy and value iteration, and then expands the size of the grid world. Ex. 21.7 asks for an exploring reinforcement learning agent tat uses direct utility estimation (implemented as table and function approximator) for a few increasingly complex worlds.

The second stage is to map the Quake work into a grid world and perform off-line learning. This is partly an exercise in abstraction and partly to produce a policy that should be approximately right in a real world setting (as in the RoboCup work of Asada and Noda, and many since [21]).

In the the third stage the students probably use some type of Q-learning.

The first (so far only) time this project was given, students evidently had some problem translating the text’s pseudocode into real programs. We think. This year the assignment (through the code and writeup of Bo Hu) provides a substantial head start with implementations of Markov Decision Processes, a uniform file format to describe the MDP, and some of the more basic learning algorithms.

Despite the initial difficulties, some student teams did quite well (Figs 4 ,5, 6).

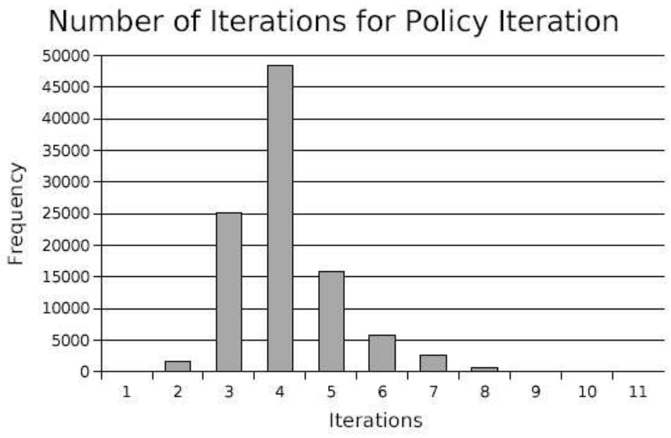


Figure 4: Policy iteration results.

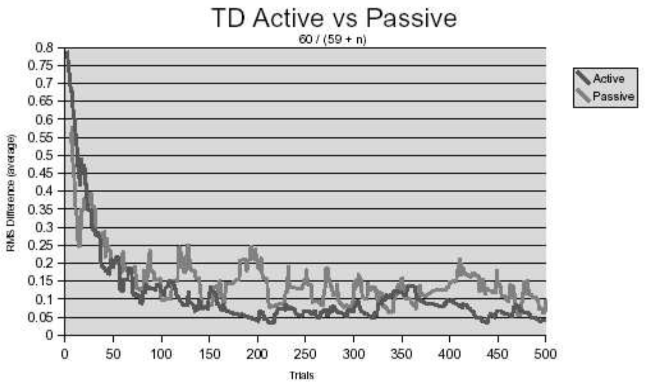


Figure 5: Active and passive temporal difference learning results.

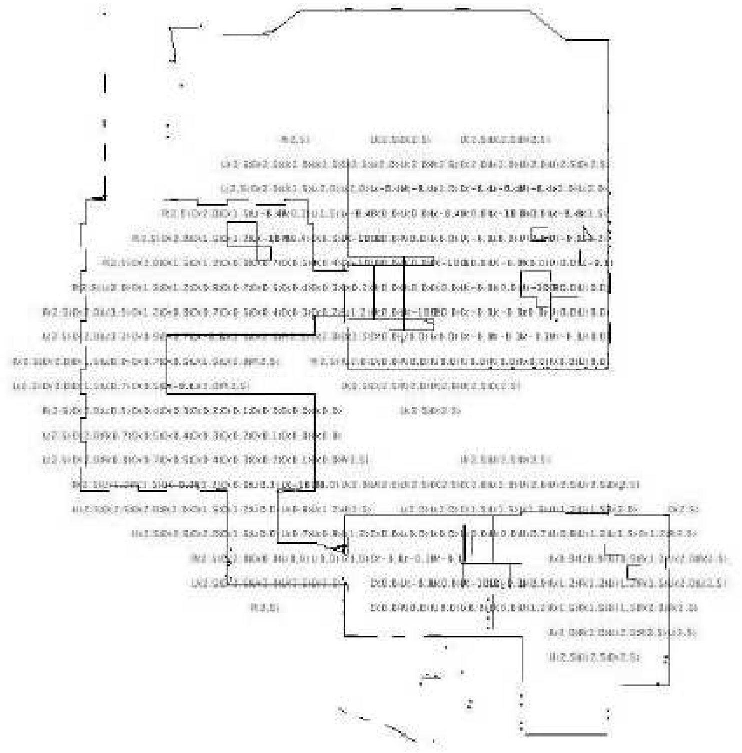


Figure 6: Learned policy for a Quake level.

Natural Language Understanding

As with the machine learning project, this was the debut of this NLU project, which was inspired by the existence of the Quagent world. This project is a Quagent version of the text's idea in Chapter 22 about ordering the agent in the Wumpus world around, possibly asking it questions, and perhaps having it respond in natural language to things it sees or experiences [58].

All the NLU processing (parsing, semantic interpretation, generation) takes place at the Quagent level, not the bot level. So the Quagent's NLU-understanding code will take as input NL sentences and produces as output quagent-protocol orders for the bot(s). Likewise the quagent program may want to turn quagent perceptions (which it may have to ask the bot for) into natural language.

It is quite possible to hack together something that works pretty well but that does not demonstrate knowledge of current NLU theory and practice. On the other hand, much of NLU theory and practice is about fairly complex issues of language in which the context is not as constrained as ours. Students thus should exhibit as much NLU expertise as possible while doing the most sophisticated an general (i.e. impressive) communication possible.

We ask for a formal grammar for the interface, a parsing algorithm, and a way to hook the parse to semantics that are used to generate the protocol messages to the bot. Students can choose to explore specific topics like anaphora, pronoun reference, or dialog (quagent asking for clarification, say).

As well as the text, we provide chapters from James Allen's text [16] on chart parsing and (Brown's current prejudice for a good mechanism for this project) semantic grammars and recursive descent parsers.

In the event, some students did quite well with downloadable software for parsing and grammars.

Computer Vision

This project simply puts some classic computer vision problems into the Quagent world. Students are encouraged to look beyond the text for background literature. Quagent bots can be given a "camera" at eye-level that allows them to look around in their world (Sec [14]).

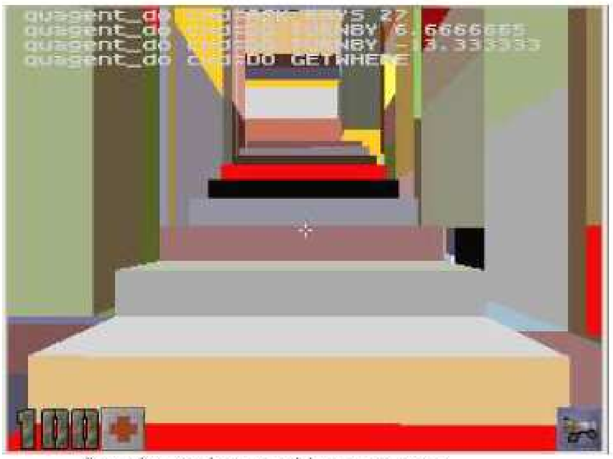
Some students overcame the menacing, ill-lit Quake II conditions by creating their own environments (Figs. 7 8).

Segmentation: Declare what objects (or parts of objects) are "salient" and use vision to separate their image evidence from the rest of the image. E.g. if edges and corners are important say to make a line drawing of your environment, find lines. Or find regions that may correspond to objects or meaningful parts of objects. The "image" might be a depth map. One could implement stereo.

Results of student experiments with edge-finders are shown in (Figs. 9, 10).

Object Recognition

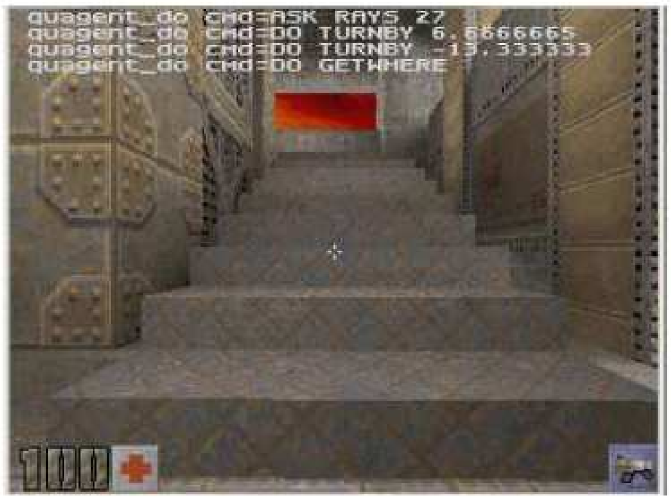
Simplify the segmentation problem somehow (normalize inputs, have predictable backgrounds...). Then tackle object recognition. Tell one bot from another, or one item from another. Our suggestion



```
quagent_do cmd=ASK RAYS 27
quagent_do cmd=DO TURNBY 6.6666665
quagent_do cmd=DO TURNBY -13.333333
quagent_do cmd=DO GETWHERE
```

Sample raw image with out textures

Figure 7: Raw, untextured image.



```
quagent_do cmd=ASK RAYS 27
quagent_do cmd=DO TURNBY 6.6666665
quagent_do cmd=DO TURNBY -13.333333
quagent_do cmd=DO GETWHERE
```

Sample image with textures

Figure 8: Textured image.

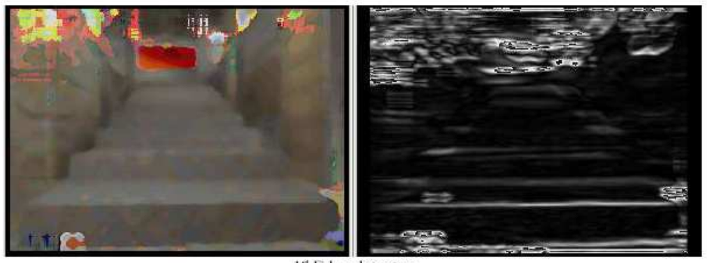


Figure 9: Edge finder 1.

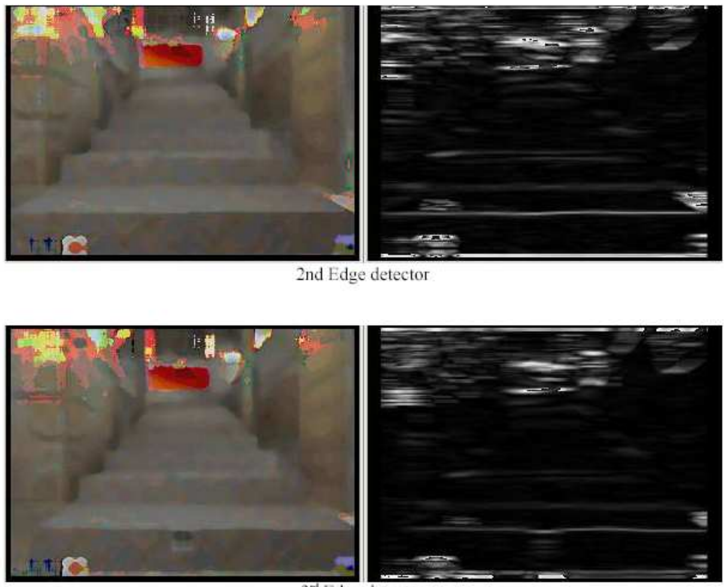


Figure 10: Edge finders 2 and 3.

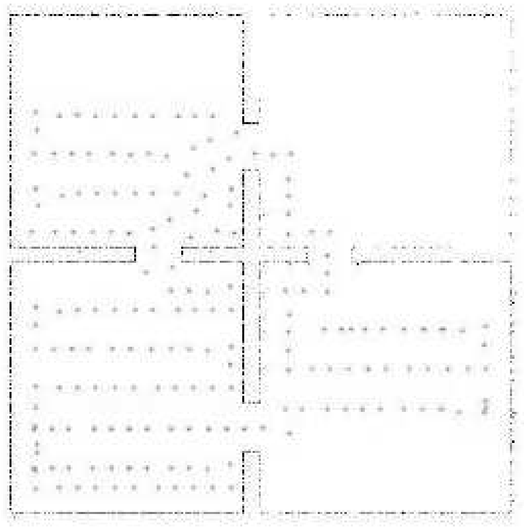


Figure 11: A map made using vision.

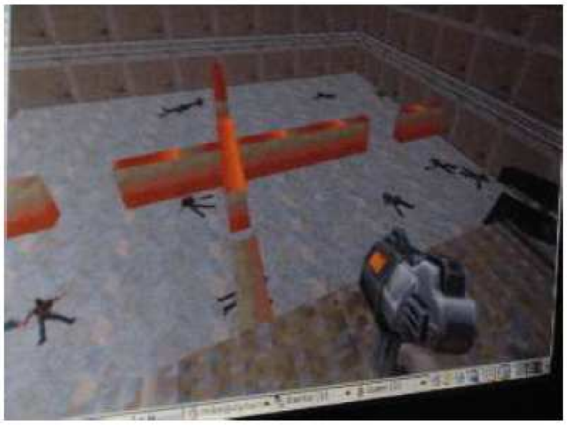
is to use a classic clustering approach or a decision tree: make visual tests of increasing difficulty (and presumably decreasing accuracy) until you're sure enough. The brave or foolhardy might want to try a neural net to implement clustering. Interesting issues here are: what features to find and how to find them, robustness of classification in the face of scale, orientation, lighting changes and obscuration. And of course how many different objects you can recognize.

Tracking: Track another object or bot. Object could be something static while observer is moving, or object could move, or both could move. This is a good place to motivate Kalman filters (Chapter 15 of the text [58]).

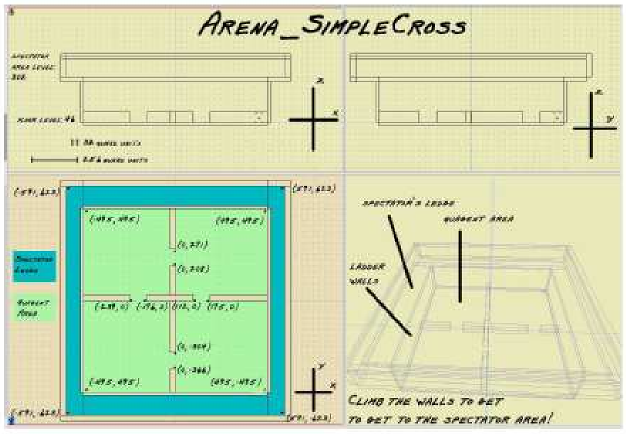
Mapping

This is This is a practical and vision-like problem. The RAYs facility simulates ring of infrared range-finders. Real IR sensors are very noisy, so one can add noise and dropout, limit sensor range, etc. For mapping, the problem is to take noisy data that should originate from walls and grouping it together into line segments so you can draw an accurate map. For this, one favorite tool computer vision classes is the Hough Transform.

We don't really expect students to write their own low-level vision code, so we provide several options for source code (including filtering code, home-grown vision code, and a pointer to the Intel Open Source Computer Vision Library).



“Hey Mike, spawn another one!”



Our environment

Figure 12: View of the situation and ground truth: some bots have expired of old age while exploring.

4.3 Objectives, Outcomes, and Evaluation

Just kidding. The relevant course has only been taught once with the quagent exercises, and its entire methodology was revolutionarily changed (from standard lectures to a “problem-based learning” format) at the same time. There are several variables changing instantaneously and simultaneously, and a historical comparison is impossible since the graduate TA (who changes every year) traditionally grades the programming projects.

The instructor (Brown) asked several explicit questions about the efficacy of quagent-based learning on the Student Course Opinion Questionnaire (SOCQ), but these particular answers were either not forthcoming or lost in the College bureaucracy. Anecdotally students seemed to enjoy the Quake and quagent world, though there were many “Quagent system is buggy” gripes on the SOCQs. The authors in general, including a serious summer-long user (Barnum,) as well as the TAs for the course, feel this complaint has little merit. In our experience instilling confidence in the system instills confidence in the students, and with time we expect more of both.

5 UR-Quake (David Costello)

UR-Quake is a modified version of Id Software’s Quake II. Dynamically, Quake II can be divided into a server part and a client part. The client part is in charge of graphics and the user controls (keyboard, mouse, game pad, etc.). The server updates the world states and feeds this information back to the client for it to redraw the graphics accordingly. Statically, Quake II consists of an executable and two DLLs (dynamical link library). The executable (`quake2`) has both the server and client code and is generally referred as the *game engine*. A video refresh library (`ref_softx.so`) provides the game engine basic graphics abilities. The game itself, containing various monsters including our quagents, is in `gamei386.so`. This arrangement enables users to load different games using a single executable. To load `gamei386.so` in current directory, e.g., we issue

```
quake2
```

To run the game, we also need to set the environment variables QUAKE2_BASE and QUAKE2_CONF to tell where the map is and a place to store configurations. The script `/u/cs242/quake/bin/run-quake` does just that.

A book on Mod programming in Quake III [39] gave useful background and general “cultural” technical background. We found useful a document detailing Quake II calling trees (function nesting) for several of the main control loops [33]. Also a very helpful and comprehensive look under the hood is [44].

5.1 Original Goals and Challenges

The project began for me essentially as a feasibility study. The question? Could the Quake game engine be used as a stand-in ’world simulator’ to support development and research of ’intelligent’ agents? The idea was that the Quake engine handled all the nasty details of collision detection and environment management so researchers could concentrate on exploring agent behaviors and interactions within a rich ’world’ space. If an object or entity bumps into a wall or falls in a pit of lava, the engine knows how to manage the physics of the subsequent consequences.

For our purposes there were two major requirements that needed to be satisfied for us to seriously consider Quake as our base platform.

1. The ”agent” code should build and run under the Linux Operating System regardless of where the actual game server was running.
2. There must exist a mechanism to build ”agents” (or at least their smarts) external to the game itself so that external resources of the agents would not have to be linked directly into the game itself.

Of course there were many more specifications afterwards but these were the main points of interest at the onset. My task was to answer these questions definitively and provide a small proof-of-concept demo. Requirement 1 just made things much easier. Linux is our primary development platform and connecting to other home grown resources such as TRIPS or computer vision applications could later be facilitated without excessive porting. Requirement 2 would allow agent code to chug away outside of the game perhaps communicating with other agents and then asynchronously agents could communicate with the game and entity representations therein.

We knew that Id software had released source code for Quake III Arena as well as Quake II and Quake I. Quake III Arena, being the latest of the series, seemed the most promising at first. It provided superior graphics to the previous two and advertised a more sophisticated internal AI for the game's bots. There also seemed to be a lot more programmer activity in terms of third party ’mods’ associated with Quake III Arena so it stood to reason that would be a good place to start. Id software does not provide a game engine API document for any of its open source code releases so you are on your own to determine the architecture by using online references in the gaming community and any possible books on the subject. The gaming community websites contain gobs of disconnected information provided by enthusiasts. The trouble with much of it is that a lot of it is in ”gamer speak” and much of that is not of a technical programming nature. Also considering that Quake III was originally released in the late 1990's (1999 and Quake II before that in 1997), many promising links to tutorials and documentation on the Internet have simply become dead links. And furthermore, although the Quake III game engine was revolutionary and is used as the base platform for many contemporary game titles, the current year was 2003. In computer game development years a five year old game release (or earlier) may as well have been 1909. The time gap is important not in terms of the revolutionary engine itself but in terms of contemporary gaming, access to shrink wrapped copies of the retail game, and ongoing game mod development. The Id software website idsoftware.com offered the source code to the game but no API documentation or code references. Planetquake.com, which is THE quake enthusiast headquarters on the Internet, proved a useful resource for finding mod programming information.

One of the first things you realize immediately is that the game was originally designed for the Microsoft Windows operating system and the source code provided by Id was configured to be compiled and run on the MS Windows platform. They helpfully provide a Visual Studio project file that does the right thing on a Windows workstation. I found some folks who had taken a crack at constructing make files for Linux but none of the distributions I tried actually successfully built a binary on our equipment, which at the time was a fairly standard installation of Red Hat Linux 9 with a gcc 3.2 compiler.

The other thing that became immediately evident is that John Carmack at Id software had NOT released the entire Quake III game to GPL open source. The client code was available to facilitate building game modifications (mods) but the game engine (server) code was kept proprietary. This was due to the fact that Id software was still selling the game engine to commercial game developers. Purchasing the game engine from Id was cost prohibitive and very restrictive. It meant you needed a commercial copy of Quake III Arena to program any game mods. You also needed to run the game (the server) on a Windows machine. Another gotcha is the fact that the data and graphics files (pak files) for the game are NOT available with the source code and are in fact not available via a GPL license. So without a retail copy of the game you cannot proceed. I decided to pursue things further anyway since I had access to a windows machine and really wanted to explore how limiting it would be to not be able to touch the engine code itself. Trying to find a copy of a game that old proved difficult. The computer stores don't carry titles that old nor do online retailers. Eventually I found a used copy of both Quake III Arena and Quake II from individuals via Amazon.com "used and for sale" section. Interestingly enough since the release of Id software's latest game Doom3 in August of 2004 a quick search via Amazon.com reveals a significant number of used copies of Quake III and Quake II. Many, many more than I found available at the time I started looking. I believe the interest in the new game has rekindled a niche market for the "old school" versions. *This means it will be relatively easy to duplicate all of our system, simply by buying the game to get what we cannot distribute.*

Thanks largely to a book by Shawn Holmes called *MOD Programming in Quake III Arena* [39] I was able to begin to understand the Quake model and successfully created some small game mods under windows to begin to explore some possibilities. Under windows I used `Q3A_TA_GameSource_127.exe` source code and applied the `pointrelease_132.exe` to the binary game installation (both available from idsoftware.com). I did things like changing the properties of a rocket so as to make it fly so slow you could fire it and then run in front of it to watch it explode close up. It was certainly not intelligence code but it helped with understanding how the game loop and event call-backs work. With the Quake III Arena source code downloaded from Id software (Q3A-129h-source was used under Linux) I did actually successfully build the client portion of the game under Linux without much trouble. But I wasn't able to obtain a Linux version of the binary game server to link to. I also explored the possibility of running the game server under windows and connecting client bots remotely. With the server locked into Windows and the networking code hidden it was going to be difficult at best to accomplish writing autonomous client bots that connected from Linux. I finally gave up trying to figure out the network protocol. So I ultimately opted to step further into the past and acquire the Quake II source which included full source code for the entire game including the engine server. The version our work is based on is 3.21 (`quake2-3.21-id.zip` from idsoftware.com)

The API, which you literally have to learn from the source code comments and word of mouth on the gamer boards, is slightly different in the older Quake II version but the basic architecture is similar so the basic feel of things I learned from the Quake III experiments was very valuable. Although Linux OS code was included in the source bundle, it would not build under Linux and it still required a commercial copy of the game for the data and graphics files contained in "pak" files. Further complicating things with Quake II is that the pak files are compressed with a proprietary compression unlike the Quake III pak files that are simply zipped and therefore can be unpacked with any zip tool. The Quake II pak files require a windows only decompression viewer tool (I used pakexplorer). Map building tools are also available only for use on the Windows platform. Once I successfully constructed a Makefile that could build the game under Linux and copied over the data (PAK) files from the retail Windows CDROM I had a complete working version of the game inside Linux.

5.2 QuakeII Bots and Mod Programming 101

The Quake II server comprises an executable and a game Shared Object or SO (or DLL Dynamic Link Library under windows) . The game SO is responsible for the AI of the monsters, weapons, bots and other game play.

The quake2 executable also contains a client part, which is connected to the server over a network, or locally if on the same machine. It is the server that dictates the events and controls the entities in the Quake II world. Everything conceived by the server is conveyed as network messages to the clients. They only have to render and tell the server what the connected player is up to (rendering may include some asynchronous interpolation of graphic details of fast events like explosions.)

Quake II is also accompanied by PAK-files. A PAK-file contains one or several maps, which define the Quake II world. They also contain information about where entities, like monsters, are placed (spawned) in the world.

The world is divided into static objects like walls, floor or sky and entities that are movable, like players, monsters, and rockets. The server maintains a large struct called edict_t that holds the representation of the world. The game shared object extends this struct to add attributes and functionality to the game. This is one place where game mod's can change the play and functionality of the game. By creating a new game shared object and loading it (swapping it for the regular game.so) with the server the "mod" is activated for play.

All entities in the game utilize a "think" function (which is provided by the edict_t struct) to execute calculations or behavior. The server loop, which is single threaded, provides each entity a slice of time to execute instructions from its current "think' function. The function pointer is changed to the function needed next by the entity and a timer variable is set for its next execution. For example a rocket in flight upon being notified that it had struck something would set its "think" function to "explode function" so that its next time slice would execute the function to perform the explosion. Entities set a "nextthink" variable timer that tells when it needs processor time again for its "think" function.

The basic directory structure of the Quake II game (and source bundle), which represents each of the main modules of the complete game, is the following:

`client/` – contains client code some of which performs local operations like sound generation to reduce network latency and off load some expensive work from the server itself.

`game/` – the server. Where the monster AI and basic behavior code for the game is.

`ref_soft/` – softx rendering code

`ref_gl/` – openGL rendering code

`linux/` – OS specific code

`qcommon/` – utility code for handling commandline, files, environment, etc...

Additionally, to run the game the baseq2/ directory containing the pak files with the game's data and graphics is required. This is obtained from the retail CDROM. Quake II Source code-naming conventions: if the source file name starts with g_ game engine (server) code m_ monster code p_ player code cl_ client code Aside:

There is a lot more "AI" code inside the Quake III Arena sources. There was a significant upgrade in the ability of monsters to play the game in Quake III which is primarily a "deathmatch" style First Person Shooter. Player skills are slightly different in a deathmatch game as opposed to an explore and advance Quake II type game. The monsters in Quake II have very unsophisticated abilities. For example in Quake II the monsters navigate their way around by following invisible breadcrumb like paths. In Quake III Arena most of the AI code is contained in source files prefixed with ai_.

The game is driven by the Quake2 engine. Every 100ms it calls a RunFrame function of the game SO, which in turn calls a tree of functions for animating all the items in the world. This is called a server frame.

Every 50ms or so the Quake2 engine calls a function named ClientThink in the game SO. Actions taken by the player are introduced at that point.

The server updates the physical configuration of the world. The client part of Quake2 predicts movement in between network updates from the server. Eventually (every few frames or so 10hz?), the engine updates the client with current information, sending its real position and orientation.

To manage time slicing and event callbacks there are a number of pointers to functions defined in the edict_t struct which can be viewed inside the g_local.h file. This is where the pointer to "think" is defined as well as the "nextthink" variable. There are also function pointers to things like "touch", which gets called by the server when a collision is detected and "die" that is called when the entity expires. All spawned entities get a handle on this struct, which is usually referenced as a pointer named *self. One way to effect a change in game behavior then is to define your own function and set the function pointer in edict_t so that your function is called when an event is triggered. To get a jumpstart you can adapt code from one of the monsters. Recall monster source code files start with the prefix m_ . In the following code snippet we adapt m_soldier.c in a new file named whatever you like.

What a simple spawn function might look like:

```
void SP_Mybot(edict-t *self){
    // use the soldier data which is inside this
    // directory in the game pak file
    self->s.modelindex =
          gi.modelindex ("models/monsters/soldier/tris.md2");
    self->monsterinfo.scale = MODEL-SCALE;
     VectorSet (self->mins, -16, -16, -24);
     VectorSet (self->maxs, 16, 16, 32);
    self->movetype = MOVETYPE-STEP;
    self->solid = SOLID-BBOX;
    self->die = myfunction_die;
    self->touch = myfunction_touch;
    self->monsterinfo.walk = myfunction_walk;
     //register this new entity with the
     //game import engine (world)
    gi.linkentity (self);
    // kickoff monster animation
    self->monsterinfo.walk (self);
    walkmonster_start (self);
}
```

Here is a very simple function myfunction_touch defined to just print to the console the name of the thing my bot is bumping into.

```
void myfunction_touch(edict_t *self, edict_t *other,
     cplane_t *plane, csurface_t *surf){
         Com_Printf("my bot is touching %s\n",other->classname);
}
```

The easiest way to execute your new spawn fuction when the game is running is to attach it to a console command. The game provides a drop down prompting interface called the console that provides certain types of game feedback, provides message passing between networked players, and allows game commands to be entered and executed. There are lots of built-in commands and it is trivial to add one that calls one of your functions. The console commands are defined in g_cmds.c. To add the spawn function above to a command called "mybot" we would add the following inside the g_cmds.c file:

```
        /* one of the built in Quake commands */
      else if (Q_stricmp (cmd, "wave") == 0)
                Cmd_Wave_f (ent);
       /* Our new command.
       * Here we look for the "mybot" string on
       * the console, print a message, and call the SP_Mybot
       * function.
       */
       else if (Q_stricmp (cmd, "mybot") == 0){
            Com_Printf("spawning mybot.\n");
            SP_Mybot(ent);
        }
```

Now when the game is running we can drop down the console (use the " ' " key) and type mybot to see the new code work.

Comment on Quake II Animations

The Quake II architecture mechanism for performing animations uses two data structures. mframe_t is essentially a list of a series of small functions to be called in a loop, and mmove_t contains instructions for which graphic frames to display during the loop. You can see examples inside the monster source files. These graphical pictures are the kind of thing you would create and then install in your own PAK file if you were creating a new game with new characters and objects. Lacking a graphical artist on the team, our initial experiments use the stock character graphics.

Graphics Rendering

An entire paper could be written about the graphical rendering of Quake II but the only thing I am going to mention is the subsystem we used under Linux. Quake II supports both OpenGL and softx rendering under Linux running an X server. Although my Windows tests had no problem building and running the OpenGL rendering subsystem I did not have any luck with it under Linux. There are a couple of issues. First, we did not have a licensed OpenGL library available. We did however have the open source MESA library available which is usually a fairly reliable substitute. Admittedly I did not spend a tremendously large amount of time trying to get Quake II to work with MESA but the time I did spend did not prove successful. The second issue is that presumably if you run Quake II Linux with the OpenGL rendering subsystem the game must be run as root (superuser) to perform the screen writes. There may be some workarounds for this but this was not an attractive restriction considering we wanted to provide a platform for students to develop agent code projects using our framework. There may also be a video card driver issue for maximizing the performance under Linux but I don't think I got that far so I can't speak to that. Once I got the softx renderer working, and found it to perform quite adequately with our hardware, I left the OpenGL as a TODO. That said, this issue will be readdressed so a comparison can be made to decide if there is a significant benefit to the OpenGL rendering under Linux.

5.3 Experimental Bot

One of our first experiments with game modifications was to create a mod (davebot) that hooks up a listening port in the game. This connection spawns a custom bot and opens a communications channel to the outside world where simple text commands are typed in to direct some bot actions. The incoming connection does some simple parsing to determine the directive and calls the appropriate bot command function. This was the beginning of development a simple text protocol into Quake II that allows external resources to work and then direct bots inside Quake (Section 7). Here is the pseudo code that formed the basis of the davebot demo:

```
davebot mainthink() pseudo-code:

[after spawn to setup communication channel for
 our external ai  text protocol.]

if (!socketInitialized){
   connect server socket and listen for connections
   (socket, bind, listen)
}
if (!socketConnected){
   nonblocking accept
   if (EAGAIN){
       nothing ready yet.
   }else{
       sock is connected
   }
} else {
  readSocket()
  parse command
  do action
}
```

5.4 Quagent Architecture

The quagent demo code and bot protocol grew out of the earlier proof of concept experiments into a kind of specialized bot proxy server (Fig. 13).

5.5 Quake II versus Quake III Arena

Quake II is fully GPL licensed and free for download. (except for PAK data)

PROS
- full engine source code included
- networking bits included

CONS
- graphics look clunkier
- builtin monster ai is very simplistic, we didn't care about this since we were not trying to actually play the game.
- OpenGL requires root under Linux
- retail version required for pak (data) files
- Only map editors we found run on Windows only

Quake 3 part of the game source is available to create dll's or .so's to create game mods.

PROS
- game AI is smarter and fuller featured (don't follow crumb trails)
- current, more state of the art engine (graphics, physics, etc...)
- the downloaded source for dll's and .so's does compile under linux and windows
- PAK files are compressed in simple zip format making them accessible without a special windows only tool

CONS
- full engine source not available and that means no networking code
- retail version required for main game binary and pak (data) files
- a lot more resources for windows platform than for linux/unix as would be expected.
- network protocol is not available from Id

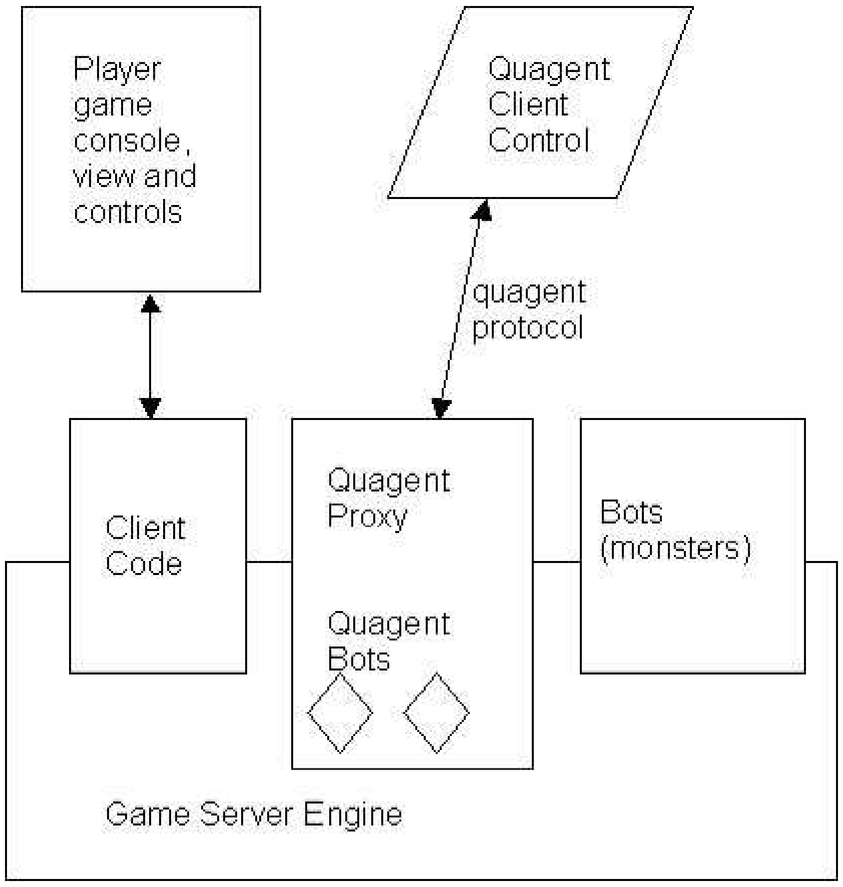


Figure 13: Simplistic Visualization of Quagent Architecture

6 The Quagent World

6.1 Quagents and Bots

In our ontology, the bots in the Quake II world are strictly hardware. In a real game, they have ”think” functions that may or not have ”mod” capabilities can be modified by the players. In Quake II, as we have seen, the bots come with a certain amount of innate “intelligence” (found in the *g_ai* files), which we have ruthlessly stripped out.

For us for now, they are commandable with a finite set of commands and they have certain reflexes that prompt them to tell you about certain events. All the intelligence therefore is off-board, outside the game, in user-written programs (in Java, Perl, Lisp, C, C++, ...). We can communicate with the game and *vice versa* (again, for now) only through a protocol. A program sends and receives strings with keywords and parameters (and it can receive binary data as well). The Quagent is thus the combination of the bot in the game and the user-written controlling software.

We can spawn multiple bots. Each one will have its own ”private line” through which you communicate to it, so users will not get them mixed up.

6.2 Objects in The Quagent’s World: Quagent Ontology

The entities in the Quagent’s world may change as we add more functionality. The issue is the Quagent’s state, perceptions, actions, and what are the interesting objects in its world.

- Time: the quagent’s world advances in 1/10 second intervals called ”ticks”.
- The Client (the first-person shooter). The game still has this character, and by default you see through his eyes. He can move around, interact with bots, presumably he can still shoot things, etc. We have disabled his ”auto pickup” property so he can’t intentionally or inadvertently change the bots’ world too much. He raises a few interesting possibilities for agents dealing with dynamic and uncertain worlds!
- Walls and windows. These are things the quagent cannot walk through.
- Items: box, gold, tofu, battery, data, kryptonite, head. You can spawn these things in locations you specify.
- Other quagents. You can spawn more than one quagent and control your bots with different controllers.

6.3 Quagent State

Quagents have an internal state influenced by what happens between it and the world.

- Position (X,Y,Z)
- Orientation (Roll, Pitch, Yaw) (most importantly for us, the ”yaw” is the robot’s ”current direction”, and most commands and reports are in coordinates that take the current (X,Y,Z) as the origin and the current direction as 0 degrees.

- Velocity
- Inventory (List of items in quagent's possession)
- Health
- Wealth
- Wisdom
- Energy
- Age

6.4 Quagent Physics and Biology

Users can easily discover how the game works, looks, and feels: there are something like normal physical laws running the show, but there are many tricks being used so some information you think should be available is either not there or hard to get. The following laws define special ways our Quagents react to things in the world.

- A bot is spawned upon connection to the game. Often this happens in the bot constructor function.
- Disconnecting kills the bot. This may be useful if the ”laws of nature” you program include fatal consequences besides the built-in ones.
- All items but boxes can be walked over; boxes block walking.
- All items can be picked up.
- Picking up (Dropping) Tofu increases (decreases) Health.
- Picking up (Dropping) Gold increases (decreases) Wealth.
- Picking up (Dropping) Data increases (decreases) Wisdom.
- Picking up (Dropping) Battery increases (decreases) Energy.
- Picking up (Dropping) Boxes just means bot has a(nother) (one less) box.
- Picking up Kryptonite means energy decreases by the same amount as if it were within the ”near” range (see below).
- Increased Wisdom increases perceptual abilities. In particular the maximum radius for the RADIUS command and the number of rays allowed in the RAYS command go up. If this linkage is irrelevant to a particular quagent scenario and experiments, the initial wisdom value can be SET to or above the value of High_Wisdom.
- Age increases every tick. When lifetime exceeded, quagent dies.

- Energy decreases by a small amount at every tick. When energy goes to zero, bot stops moving. Everything else still works, including Pickup, so another bot could give it batteries.
- When Health goes to zero the bot dies.
- If Kryptonite is within a ”far” range it further decreases bot energy by a little at every tick. If it is within a ”near” range it decreases bot energy by more at every tick.

These are a couple of dozen parameters set by the system and are not settable by the user governing the physical constants in the Quagent world.

6.5 Configuration Files

There are two sorts of configuration files. There is at most one of the first type, which sets up the world and is called when the game is launched. It populates the environment with items in selected locations select.

Items can be spawned into a quake world from the console or can be controlled by a configuration file. The file should be called `quagent_world_config.dat`

It has a simple format: it has one line per item you want spawned and each line looks like

[item name] [x] [y] [z]

Where [item name] is one of BOX, TOFU, BATTERY, GOLD, DATA, KRYPTONITE, and [x] etc are strings to be interpreted as numbers giving (x,y,z) absolute Quake coordinates. Now this means one needs a map of the current room in order to know where things are, and so things will not be spawned inside walls or out of the room entirely.

With no configuration file a user gets the standard items the game spawns.

The second type of configuration file configure the bots.

Each time a bot is spawned its individual config file may be supplied. This lets one do experiments with bots of different capabilities. The bot configuration file has lines allowing initialization of certain bot state levels and thresholds. They usually look like

[variable] [value]

Where [variable] is one of LIFETIME,
INITIALWISDOM,
INITIALENERGY,
INITIALWEALTH,
INITIALHEALTH,
ENERGYLOWTHRESHOLD,
AGEHIGHTHRESHOLD,
and [value] is a string to be interpreted as a number.

An important one is slightly different, taking three values giving (x,y,z) location of bot.

INITIALLOCATION [x] [y] [z]

With no configuration file a bot is spawned in a default room (in fact the first room in the game) at a default spot with default values.

Here is a sample configuration file:

```
   quagent  type CB lifetime 5000 initialenergy 1000
quagent  type Lane lifetime 6000 initialenergy 2000
quagent  type Randal lifetime 3000 initialenergy 3000
tofu 128 -236 24.03
data 150 -200 55
battery 50 50 100
gold 300 -200 30
kryptonite 100 50 100
```

6.6 Quagent Communication and Interaction

As of this writing (09/04) communication between quagents (not bots) is easy if they are all running in the same program. It may be a little harder (and non-uniform) if they are not. Right now we are not using game facilities to do communications since they are set up for client to client communication.

As of now, think about the quagent programs, the controllers, communicating, not the bots communicating. It's like the language and reasoning centers are up in the controller. So one quagent can broadcast, or target another particular quagent, or all quagents within some distance, with ”messages” that can be interpreted by the other quagent (controller.) Thus stylized behaviours, or sequences of actions, can be scripted: Quagent A can be low on GOLD but have a lot of HEADs, so it could broadcast ”OPENFORBUSINESS”, whereupon B could decide to swap GOLD for a HEAD, and so the controllers would cooperate by running off the necessary walk-over-to, drop gold, drop head, pickup head, pickup gold actions.

Likewise there could be a STEAL interaction that could be built same way, failing if the desired item is not held, OR one could implement the capability of A inquiring of B what its inventory is. Again the whole transaction would be mediated by the quagent code, not the bot code.

A good quagent-quagent communications protocol would be a nice contribution. Barnum has implemented one and it may appear in the disseminated code, web pages, and possibly in this TR soon.

6.7 Designing Environments

Most students ultimately like to design an environment that has the features needed by their application and avoids distracting unneeded features. To create such a custom environment for your Quagents you use a *level editor*.

The two main editors that people mainly use are both for Windows, however the map files they create work in Linux too. A good one is Worldcraft 1.6a [12], which is supposed to be the easiest to use. Some other people use the Quark, or the Quake Army Knife [11].

There are rumors about level editors for Linux, but haven't found anything solid. The worldcraft site has tutorials, and it is possible to make maps fairly easily using it. Sometimes textures have translation problems between the level editing and running in Linux. The only fix we know is just to pick different textures.

6.8 Playing God

Clearly some of the basic laws of how bots interact with the world could easily be enforced by the quagent controller. For instance, to enforce weakening around kryptonite a user could write the same thing we did...check for kryptonite and decrement some ”health” variable accordingly. In fact there is no reason users cannot build complex phobias, social behaviors, health quirks, prioritized but conflicting motivations, whatever! The causes, effects, and interactions can all be overseen by the quagent controller. This would be just another layer of simulation overlaid on the existing minimal bot capabilities. So don’t be frustrated if the bar bots seem to have a minimal inner life (sort of a physical body and maybe an id). You can build the ego, superego, etc. in the controller.

Users can build and extend many quagent functionalities, and they should not feel constrained by the limited amount of on-board ”wired-in” bot characteristics we supply. For example, for research purposes Barnum eliminated many of the Quagent biological laws (Sections 3 and 9).

7 Quagent Protocol

In this and some succeeding sections, URLs are in [square brackets], and the “...” stands for “http://www.cs.rochester.edu/u/brown/”. Bots are controlled by and communicate with quagent controllers using the [.../242/docs/quake_protocol.html] Quagent Protocol (described below). For how to use the protocol in code, see [.../242/docs/quake_code.html]. The latter reference describes the quagent configuration file that set ups the bot and its environment (Section 6.5).

The Quake file `game/g_cmds.c` implements console commands, which allow direct interaction with the game. Consult game documentation or the file to see what is possible.

The Quagent protocol is based on our experience with intelligent interactive computer agents. The controller can send the bot commands or ask it things. The bot confirms command receipt, sends error messages, or sends back data in response. The bot can also volunteer facts to you about events it senses.

The general form of protocol messages is:// From controller to bot: // DO [Command] [Parameters]// ASK [Query] [Parameters]// From to bot to controller: // OK ([Echo]) (echos for positive confirmation, as in a submarine movie) // ERR ([Echo]) [Error Description] (can’t begin to do command).// TELL [Event] [Parameters] (Asynchronous from bot: volunteers information).

In the following, + and * are used as usual in regular expressions. UPPER-CASE strings are keywords. [lower-case-id]s represent strings that are parsed by scanf as an integer or real number. [Echo] is shorthand a copy of the command string in the response. [Error-cause] is shorthand for a helpful error description. The Response// OK ([Echo]) or ERR ([Echo]) [Error-cause], in which the bot agrees to try to start the command, or to perform it, or says why not, is called the *Standard* response. Note the parentheses around the echoed response.

[Item-Name] is one of the strings BOX, GOLD, TOFU, BATTERY, DATA, KRYPTONITE, HEAD.

7.1 Action Commands

Command: WALKBY [distance] (e.g. WALK 20.0). Desire Bot to start walking in current direction. Bot will, at best, start walking in the right direction and silently stop when distance is reached. ¡b¿Surprisea¡/b¿ The bot is really a bounding box and some graphics that change, often in a cycle, when it is doing something, for example walking. The ”realistic” look requires that the bot move unequal forward distances between the various images that depict its action. You can see sample code in ”monster” source files like .../game/m_gunner.c, which define the graphics, speed of motion, step sizes, etc. for the different quake denizens. Thus your bots take ”different sized steps” during their walk cycle. Thus they only approximate the distance in the walkby command, and if you are counting steps (ticks) or something to compute your own distances, you can be surprised.

Response: Standard.

Command: RUNBY (Similar to WALKBY)

Similar commands:

- STAND. Bot stops moving.
- TURNBY [angle]. Angle is in degrees, + (left turn) or - (right turn). Changes current yaw (orientation). Usually performed when bot is standing, maybe no reason not to issue while moving.
- PICKUP [Item-Name]. Immediately picks up one of the named items if bot is within the ¡Pickup Radius¿ of the item. Error occurs if item is not close enough.
- DROP [Item-Name]. Immediately puts down one of the named items in the bot's inventory. Error occurs if bot not holding named item.

7.2 Perception

- ASK RADIUS [distance]. What items are within the given radius? (There is a system-imposed max-radius).

  Response: ERR (Echo) [useful_description] or // OK ([Echo]) [number_of_objects] ([object name] [relative_position])*//, where [relative_position] is an (x,y,z) three-vector of relative distances. Example: OK (ASK radius 100.0) 2 GOLD -20.0 30.0 0 Sandhya -320 -100 0
- ASK RAYS [number_of_rays]. What entities are surrounding bot in some set of directions. If number is one, ray's direction is in bot's current direction. The command shoots out [number_of_rays] evenly distributed on a circle around the robot. // Response: as for RADIUS.// Example: ASK RAYS 10// OK (ASK RAYS 2) 1 world_spawn 315.0 277.1 0.0 2 TOFU 200 100 0 // Response: ERR [useful_description] or // OK ([Echo]) ([ray_number] [object_name] [relative_position])+ // relative_position is as in the Ask Radius command. The ”world_spawn” entity is a wall or other game structure.
- ASK PING (a high-resolution version of RAYS).
- CAMERAON. Normally the terminal shows the view from the (first-person) client. This puts the camera on the bot.

- CAMERAOFF. Puts camera on client.
- LOOK. Uploads image buffer to the quagent code, presumably the world as seen from the bot's point of view.

7.3 Proprioception and Asynchronous Condition Messages

It is useful to be able to query the bot's internal state ("get" commands) and to set certain bot parameters for experimental purposes ("set") commands.

- GetWhere. Where is bot, how oriented, moving how fast? these values must be in world coordinates to be meaningful. // Response: ERR ([Echo]) [useful_description] or // OK ([Echo]) [World_State], where [World_state] is a vector of coordinates and a velocity: (world_x, world_y, world_z, roll, pitch, yaw, velocity).// Currently (27 Jan 04) the velocity is returned as a constant 1. It's tricky, not obvious how we can set or get it. If there's a crying need, let us know through the suggestion mechanism.
- GetInventory. What is bot holding?
- GetWellbeing. How is bot doing in its life?
- SetEnergyLowThreshold [value]. Also settable in the configuration file. When energy drops to this value, bot sends a single TELL message to the controller.
- SetAgeHighhreshold [value]. Also settable in the configuration file. When rises above this value, bot sends a single TELL message to the controller.

[Condition] is a string signaling a condition either in the bot or in the world. The bot can volunteer the existence or parameters of this condition, or of an event in the world, using the TELL "response". TELL is not elicited by any DO command, but is sent unsolicited by the bot when the given condition is detected.

Command: None

Response: TELL [Condition].

The following [Condition]s generate TELLs:

- STOPPED Motion stops before desired WALK_BY distance attained.
- KRYPTONITENEAR: Kryptonite has entered the near effect range (from out of range or far range).
- KRYPTONITEFAR: Kryptonite has entered the far effect range (from near or out of range).
- KRYPTONITENOT: Kryptonite has gone outside the far effect range.
- LOWENERGY: A warning: Energy fallen below Low-threshold: paralysis looms.
- NOTLOWENERGY: Energy risen from below to above Low-threshold.

- OLDAGE: A warning: Age above threshold: death looms.
- DYING: [World_State] [Inventory] [Well_being]. Bot expiring of old age. Death rattle and dying dump.
- STALLING: [World_State] [Inventory] [Well_being]. Bot out of energy, alive but unable to move.

7.4 Personal Interaction with Quagents

We interact with quagents through the quagent protocol described above. The protocol is in ASCII, so telnet works to try it out. When the game is launched, it appears that it grabs the mouse. To move the mouse to another window to do telnet-ing, hold down the `SHIFT` key and then move the mouse. Change to another window by using the key-combination to switch windows defined by your window manager. For KDE, this is usually `ALT-TAB`. This is handy when trying agent programs.

A real session could look like this:

```
prompt: telnet localhost 33333 (Connect to a quagent).
Connected to quakeII bot (Greeting message: connected)
do walkby 100   (Issue a WALKBY command)
OK  (do walkby 100) (Quagent acknowledges by echoing the command)
do turnby 90   (Have the quagent turn by 90 degrees)
OK (do turnby 90)

TELL STOPPED 60.0045  (We let the quagent walk 100.  It has walked
   60.0045 and stopped, probably hitting a wall, triggering
    a STOPPED event.
   Note that this event is asynchronous, i.e. you don't know
   when it'll come.)

TELL KRYPTONITEFAR (We get yet another asynchrounous event that
   tells us there's a kryptonite within 200 units.  Apparently
   the consequence of the walkby).

ask radius 100 (Issue a query, what's out there?)
OK (ask radius 100) 1 player 156.5 86.9 23.9
    (Only sees the player (that's you))
```

8 Programming Issues in Creating Quagents (Bo Hu)

As we saw in Section 5, Quake II consists of three parts: the engine, the game module and the video refresh module. The three parts communicate with each other through a set of APIs. The game module exports `game_export_t` to the engine and imports `game_import_t` from the engine. There

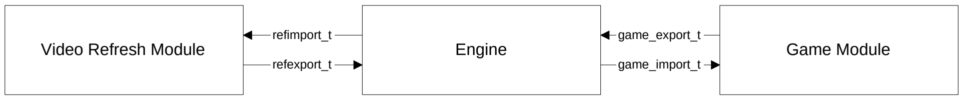


Figure 14: Relationships among the three modules of Quake II.

are two types of video refresh modules, one of which uses OpenGL and the other uses the shared memory extension of XFree86. Both types provide a set of APIs to the engine with `refexport_t` and use functionalities defined in `refimport_t` from the engine (Fig. 14).

8.1 Endowing Quagents with Vision

Visual perception is an important aspect of an intelligent agent. There are two approaches to give vision to a quagent. The hard way is to render the scene from the view point of the quagent by ourselves. This involves deep understanding of the Quake II graphics code. We take a shortcut. Noticing that the scene is rendered to the view point of the client (the first person shooter), we manage to steal the camera of the client and attach it to a quagent. We then read off the rendered image from the frame buffer. The disadvantages of this clever laziness are, firstly, we are stuck with the resolution of the game, i.e., if the game is set at 800×600, the images we obtain are of that size or a down-sampled size; secondly, there can be only one camera at a single moment. Nevertheless, these disadvantages have not shown great limitations in our research and teaching so far.

The way we implement the frame buffer capture also exhibits how to bridge the three parts of Quake II. As can be seen from Fig. 14, the video refresh module does not talk to the game module directly. So we need to augment `refexport_t` and `game_import_t` to enable the game module to access the frame buffer. Specifically, we add `GetFrameBuffer` to `refexport_t` and `game_import_t` and link the latter two during engine initialization.

8.2 Transfiguring Monsters to Quagents

Every monster in the Quake II game is of type `edict`, which records all information about a monster. To turn a monster into a quagent, we add `quagentinfo_t` to `edict`. The full content of `quagentinfo_t` is shown below.

```
typedef struct quagentinfo_s {

    // communication stuff
    int connected_socket;  /* connected client socket */
    int port;

    // states
    // health, position and direction are already in edict_t
    float energy;
    float tell_weak_threshold;
    int wealth;
    float speed;
    int age;
    int too_old;            // when age>too_old, it dies of aging.
    int tell_old_threshold;
    int wisdom;

    // all other states
    unsigned long misc_states;

    // objects it owns
    struct gitem_s *inventory[MAX_ITEMS];
    int invcount;

    // alarm mask
    unsigned long alarm_mask;

    // requested walk distance: ai_walk doesn’t keep track of distance
    // to make walk_by possible, we need this state variable.
    float desired_distance;
    float walk_distance;

    // message is a scratch space for communication, because we are using
    // strings to convey all information, we need space to store them
    // this is especially convenient for the functions to return extra
    // information to the caller
#define QUAGENT_MESSAGE_LEN 1023
    char message[QUAGENT_MESSAGE_LEN+1];

    // internal functions
    int (*trigger_alarm)(edict_t*, unsigned char alarm, ...);
    int (*die)(edict_t *);

    // what this guy can do
    int (*turn_by)(edict_t*, float yaw_angle);
    int (*walk_by)(edict_t*, float distance);
    void (*stand)(edict_t*); // don’t want to change DC’s quagent_stand

    int (*pickup)(edict_t*, char *item);
    int (*putdown)(edict_t*, char *obj);

    int (*tell_radius)(edict_t*, float radius, unsigned long mask);
    int (*tell_ray) (edict_t*, int num_rays);

    int (*camera_on)(edict_t*);
    int (*camera_off)(edict_t*);
    int (*image) (edict_t*, int *size, char **image);

    // for each property of quagent, there is a set and get function
    int (*set_walk_speed)(edict_t*, float speed);
    int (*get_walk_speed)(edict_t*);
    int (*set_alarm)(edict_t*, unsigned long alarm_mask);
    int (*get_alarm)(edict_t*);
    int (*set_age_threshold)(edict_t *, int );
    int (*set_energy_threshold)(edict_t *, float);
    // we don’t provide set_(wealth|energy|age), though
    int (*get_wealth)(edict_t*);
    int (*get_energy)(edict_t*);
    int (*get_health)(edict_t*);
    int (*get_age)(edict_t*);
    int (*get_where)(edict_t*);
    int (*get_wellbeing)(edict_t*);
    int (*get_inventory)(edict_t*);

}quagentinfo_t;
```

In an Object-Oriented terminology, a quagent is a sub-class of a monster. Basic behaviors, such as standing, walking, running, searching, firing weapons and dying, of monsters are defined in `g_ai.c`. In general, we do not need to change these behaviors. Instead, we teach these old dogs new tricks. We add new abilities to each action. For instance, the original `ai_walk()` does not count how far a monster has gone. However, a quagent needs such ability. So we augmente `ai_walk()` to keep track the distance being walked. To tell a quagent apart from a regular Quake monster, we assign a class name to every quagent. All the class names start with `bot_`. Therefore, to determine whether an `edict` is a quagent or a monster in any `ai_` functions, we can use the following construct.

```
if(strncmp(self->classname, "bot", 3) == 0) {
    mainthink();
    // do other quagent tricks
}
// original monster behavior
```

The function `mainthink()` in above snippet checks if there are commands from the communication channel (BSD socket in this case) and performs the commands if there are. The whole Quake II game works like a co-operative multi-task operating system (remember Mac OS pre-X and MS Windows pre-95?) Each `edict` does its job in allocated time slice and voluntarily release control to others. The above construct guarantees that `mainthink()` is executed when a quagent gets its slice, so the quagent can response to commands promptly.

With appropriate values filled in the `quagentinfo_t` and augmented `ai_` functions, turning a monster into a quagent becomes a mechanical operation. Each monster has its signature behavior built upon the corresponding `ai_` function. For example, how fast a monster walks and the way it walks are determined by a combination of animation frames and `ai_walk()` calls. This combination is well tuned and we do not need to change it. Each monster has a *spawn* function, which is called when the game needs to create or spawn a monster. A spawn function has the name of the form `SP_monster_`. In Object-Oriented terms, this is the constructor of a monster. It initializes all the properties of the monster. This is the function we need to extend so that we can construct a quagent. The constructor of a quagent bears the prefix `SP_quagent_`. Since a quagent is a sub-class of a monster, the constructor inherits all the content of a monster spawn function. Futhermore, we assign a class name to the quagent and fill the `quagentinfo_t` in the constructor. The quagent constructors are called when the quagent module is initialized from a configuration file 6.5.

8.3 Spawning objects

The procedure of spawning or inserting an object into the world is

1. Finding the item using `FindItem`, which is provided by the game.
2. Specifying the graphics of the item.
3. Placing the item at desired location.
4. Spawning the item using `SpawnItem`, which is provided by the game.
5. Setting the pickup function of the item to `quagent_pickupitem`.

There are assorted items in the full Quake II game pack. One of the team members even dug out an ancient head, which arguably resembles the Microsoft-you-know-whom. We build a translation table to map Quake II items to quagent items. For example, a health box is mapped to tofu. To show the loving of peace of this team and to avoid legal issues, we will make our own items. But the procedure is the same.

9 Modifying Quake for Team Coordination and Trading (Peter Barnum)

9.1 Creating Uniqueness

In URQuake, there are large groups of objects, from items to quagents to walls and doors. Although Quake has an internal representation of each that separates it from others, a basic quagent can only see each object's item type (e.g. one health box looks the same as any other.) This representation works fine as long as quagents do not need to communicate or cooperate. However, imagine the dilemma if there were three quagents of the same, and one wanted to communicate to another. It would need some way to tell its communication protocol exactly who to talk with. To allow unambiguous message sending, we gave each quagent a unique ID number.

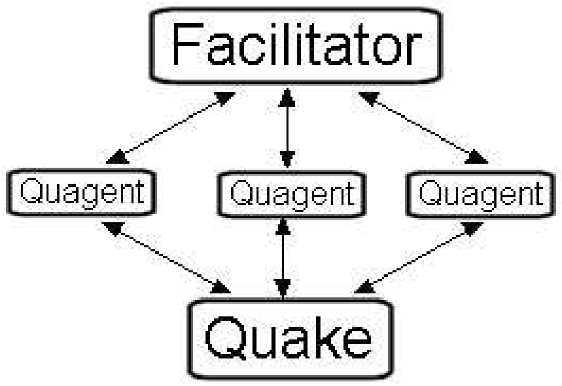


Figure 15: Facilitator

9.2 Unique Quagent IDs

Quagent rovers are made up of two parts. The first is the simulated physical rover in Quake, and the second is the controlling intelligence, which communicates with its rover over a socket connection, in the same manner as standard quagents. In our initial implementation, quagents can interact physically with each other, (mostly by getting in each other's way,) but had no way of coordinating their actions via communication. In keeping with our policies of not putting higher-level functionality, especially that related to research, directly into the Quake engine, we created an external facilitator program through which each quagent can communicate with the others.

The facilitator has socket connections to the user-written quagent controller code (Fig. 15.) If one quagent wants to talk with another, it sends the message through the facilitator, which then passes it on to the intended recipient. The facilitator simplifies quagent-quagent communication, as each only has to have a single, permanent socket connection, instead of needing to handle connections to every other quagent.

When a quagent looks around in the Quake simulator, it can see the ID numbers of other quagents. Using this ID, it can send messages to the quagent it sees, even though there is no direct link between quake and the facilitator. In Fig. 16, two bots see each other and their respective IDs. Quagent 5 wants to offer a trade to the bot it sees, so it sends the message through the facilitator. Because the IDs in Quake and the Facilitator are synchronized at start-up, message passing is unambiguous, although it has a layer of indirection.

9.3 Unique Items

Each standard Quake item is identical to every other item in its class (e.g. every tofu is the same as every other tofu.) Sometimes, however, it is desirable to have an object with properties that are subtly distinct from others, or else are modifiable dynamically as the simulation progresses.

Modification is perhaps the most useful aspect of unique items. For example, suppose there is a need for a pile of 100 gold coins. The standard Quake would need to spawn 100 separate coin items. On the other hand, with unique items, only a single item, “pile of coins,” would need to be created.

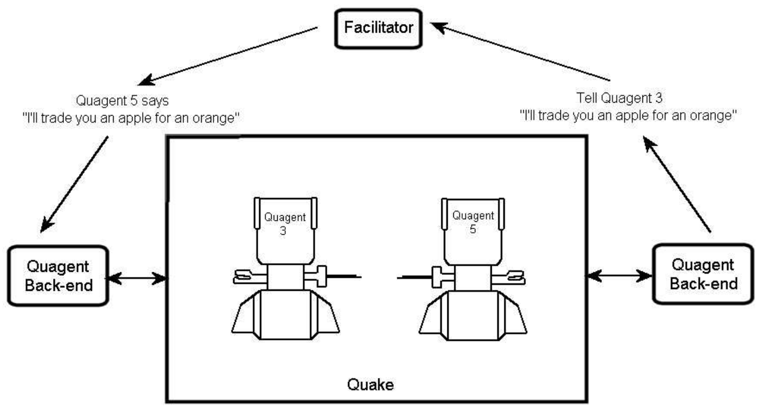

Facilitator
Quagent 5 says "I'll trade you an apple for an orange"
Tell Quagent 3 "I'll trade you an apple for an orange"
Quagent Back-end
Quagent 3
Quagent 5
Quagent Back-end
Quake

Figure 16: The Quagent world

The pile could be made so that as quagents take coins, the number of coins remaining diminishes. In this way, implementation is simplified and design becomes more intuitive.

Another useful property of unique items is that it is simple to create a large number of fairly similar items, without having to define a new item class for each one. Continuing with the gold coins example, it is trivial to make piles with from one to one thousand coins with a unique item, by modifying only a single number in the configuration file. By contrast, with standard items, a thousand item classes would have to be manually defined for the same functionality.

9.4 Trading Items

Trading items requires the use of both quagent and item IDs. Expanding on Fig. 16, two quagents use the facilitator to arrange a trade. Eventually, they mutually decide for Quagent 5 to give an apple and for Quagent 3 to give an orange in return. The actual "physical" items are stored in the Quake simulator, and the manipulation of inventories must happen there.

We modified the Quake code to add a protocol to trade items, money, and information. A trade is a transfer of any or all of the three between two quagents. Only items are handled in the actual Quake program; money and information, being new to the Quake ontology, are handled in the quagent controllers. If a trade is completed successfully, then Quake will send a message telling the quagents to transfer funds or information. It is impossible for a quagent to renege on an item trade, as it is handled directly by Quake, but money and information require good faith (or "honest" quagent controllers.) The issue of a dishonest or malfunctioning quagent is an separate problem that won't be handled here. Any number of protocols to decide on trades could be used, with no additional modification to the Quake code.

In the basic example given, the trade involves only items. Suppose now that Quagent 3 has no oranges, but instead offers to trade $50 and the location of an orange grove for the apple. Quagent 5 feels that this is fair, and they begin the trade protocol.

Quake requires that bots give permission for inventory modification, to prevent stealing. So to begin, Quagent 3 must tell Quake to allow Quagent 5 to modify its inventory to give the apple. It tells Quake a message such as: "Allow Quagent 5 to give me an apple. I agree to give $50 and

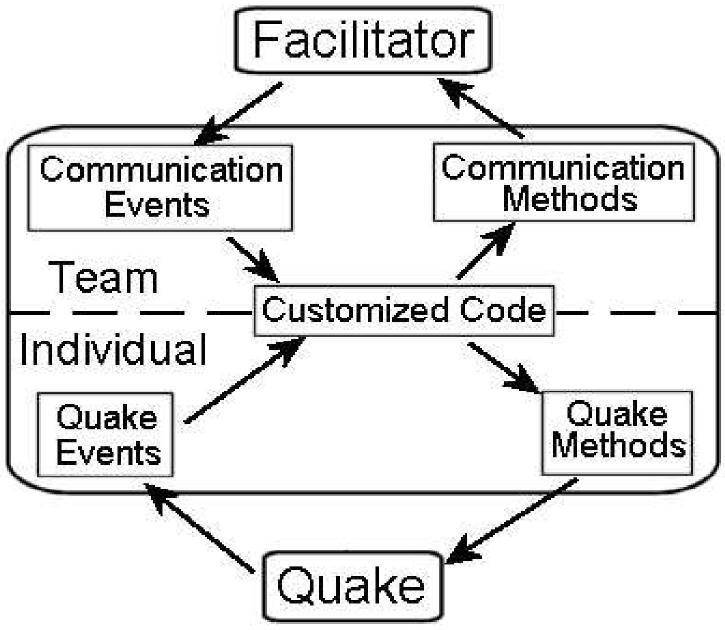


Figure 17: The TeamQuagent interface

the location of an orange grove in exchange." Quake receives this message and adds it to a list of acceptable trades. At this point, Quagent 5 can perform the actual trade. It tells Quake: "Give Quagent 3 an apple. Request $50 and the location of an orange grove in exchange." Assuming that they are in range of each other, Quake will move the apple from Quagent 5's to Quagent 3's inventory, then send a message to Quagent 3 ordering it to send the money and information. Quagent 3 will then transfer the funds and information via the facilitator, completing the trade. If for some reason, the acceptable trade and the trade attempt are not the same, then the trade will fail and nothing will be exchanged.

9.5 The TeamQuagent Package

The TeamQuagent package contains low-level routines for quagent functioning, including a Java class to extend and a separate facilitator for communication between TeamQuagents (Fig. 17.) In the picture, the relationship between different pieces of code can be seen. In the customized code section, a developer can create individualized functionality, while using the TeamQuagent package to communicate and coordinate with other Quagents and Quake. Events are collected asynchronously by the communication and quake events modules, and high-level commands can be sent with the methods modules.

A message between quagents can contain an arbitrary collection of Java Objects, wrapped with the TeamMessage class. If a quagent wants to send a message, it makes a new TeamMessage of the appropriate type, and passes it to the communication method, which sends it on to the facilitator. As an example, a text message can use the built in extension TextTeamMessage, which allows the sending of a String Object.

To interface with Quake, TeamQuagent provides a group of methods to send commands. For instance, calling the runby(int distance) method tells Quake to make the bot run. TeamQuagent creates the appropriate string in each method, and passes that to Quake.

TeamQuagent has two event queues, one for Quake and the other for messages from other quagents. While running the main control loop, a TeamQuagent periodically can check the queues and handle any events or messages that came up. A Quake message is the result of a scan or trade, or an unexpected event such as running into a wall and stopping.

By using the quagent-to-Quake and quagent-to-quagent methods, a fairly sophisticated bot can be constructed without having to get bogged down in any low-level system or networking commands. Also, the multi-thread safe nature of TeamQuagent allows for robust, real-time applications to be constructed without extra checks.

9.6 Minor Protocol Changes

Apart from changes to allow uniqueness in quagents and objects, a few extra functions were added and modified: (see Section 7 for details on the semantics of arguments.)

- Added (For use with standard or team quagents)

  1. DO RUNBY - RUNBY works just like WALKBY, except that the quagent will move faster and use different animation frames.
  2. ASK PING - PING acts in a similar way to RAYS, except that it sends out a dense wedge around a heading, rather than evenly spaced. In addition, unlike RAYS, PING will only respond if it hits a wall, passing through any other quagent or item.
  3. DO DRILL - DRILL requires the unique item Rock. Each Rock has a certain amount of two types of minerals. DRILL will remove a unit of one of the types, and create a new mineral item in the quagent's inventory.
  4. DO SETEXEMPT (0 or 1) - SETEXEMPT will make the quagent exempt from kryptonite, dying of old age, running out of energy, limited number of rays, and limited distance for RADIUS.

- Modified (Changed for use by team quagents)

  1. DO SETID # - The # in SETID should be the same as the one that the quagent will be using for communication. When SETID is used, the quagent will receive modified information from RADIUS and GETINVENTORY.
  2. ASK RADIUS - After DO SETID is sent, ask radius will respond with a list of objects, of type Quagent, Standard Item, and Unique Item. All have a name and position, but Quagents also have a bot type and ID, while Unique items have a list of properties.
  3. DO GETINVENTORY - In addition to the standard response, after DO SETID is sent, GETINVENTORY will send a TELL with a list with the names and IDs of all the inventory objects.
  4. DO ADDTRADE [with who] [Agent1 item] [Agent2 item] [Agent1 money][Agent2 money][Agent1 info][Agent2 info] - Will add an acceptable trade to the quagent's list.

5. DO GETTRADES - Returns a list of all of the pending trades that the quagent will allow.
6. DO REMOVETRADE [trade ID] - Removes a trade from the list of pending trades, so it will no longer be allowed.
7. DO TRADE [with who] [Agent1 item] [Agent2 item] [Agent1 money] [Agent2 money] [Agent1 info] [Agent2 info] - If the other quagent is in range and has agreed to the trade, then it will be completed.

A The Source Code Distribution

To recreate a full-blown Quagent system, you may need to acquire a copy of the game to get access to the proprietary graphics, monsters, and levels (See Section 5). Everything else you need is in our source code distribution.

Alternatively, you can build a system with full Quagent functionality useing the freely-available demonstration code from id Software [6]. You should get their source bundle from `http://www.idsoftware.com/games/quake/quake2/index.php?game_section=demo`, use `unzip` to extract the contents, `mv Install/Data/baseq2` to some directory of your choice, for example:

```
 cd Install/Data/
mv baseq2 /usr/local/quake/
```

When building the source code tell configure where to find it, for example:

```
./configure --with-baseq2=/usr/local/quake/baseq2
```

If you have the retail version of the game, you get more maps and resources by copying the its *baseq2* directory contents.

One of the code examples uses the production system language Jess [7]. Jess is easily, and usually freely available by download and requires an academic-style non-dissemination and not-for-profit license, also available on line.

There is more, and more up-to-date, information at our website: `http://www.cs.rochester.edu/research/quagents/running.html`.

You can easily use a level editor [12] to create a bare-bones system, with a bot that looks like, say, a sphere.

Generally you may find that different graphics systems or other components are easier or harder to bring up in your particular environment.

Our source code, documentation, examples, etc. is all available on our main Quagent web page [14].

B Programming Quagents: Example

B.1 Encapsulating the Command Interface

You might want to insulate yourself from the details of dealing with messages by providing objects and methods that encapsulate the parsing details. For example:

```
      Command cmd = new WalkCommand(20);
      this.send(cmd);
      Query query = new RadiusQuery(33.0);
      Answer answer = this.ask(query);
      if (answer.isOK()) {
          ... extract answer.value() and cast or something ...
      }
```

B.2 Drunken Master Example

To help you understand the protocol better, let's build a random-walking quagent using Java. You can use any programming language. We use Java because (1) we'll take advantage of its OO constructs and (2) it's easier to integrate with [http://herzberg.ca.sandia.gov/jess/]Jess¡/a¿. We call our random-walking quagent `DrunkenMaster`. It only does three things until it dies of aging. It reports what's around it, walks for a random distance and then turns by a random angle. Before we go directly design the `DrunkenMaster`, we can build a basic quagent, which does basic things such as sending requests to the quagent module and receiving responses from it. This layer of abstraction insulates us from the boring details of composing requests and parsing responses. The outline of our `DrunkenMaster` looks like this:

```
public class DrunkenMaster extends BasicQuagent
{
        public void Run() {

            while (alive) {
                RadiusQuery qry = new RadiusQuery(100.0f);
                send(qry);
                Response rsp\_rad = getResponse(qry);
                if(rsp\_rad.isOK())
                    ((RadiusResponse)rsp\_rad).showItems();

                double dist = Math.random() * 100.;
                WalkCommand walk\_cmd = new WalkCommand(dist);
                send(walk\_cmd);
                Response rsp\_walk = getResponse(walk\_cmd);

                double angle = Math.random() * 180.;
                TurnCommand turn\_cmd = new TurnCommand(angle);
                send(turn\_cmd);
                Response rsp\_turn = getResponse(turn\_cmd);
            }
        }
}
```

B.3 Discussion

Our `DrunkenMaster` is a subclass of `BasicQuagent`, which provides the `send()` and `getResponse()` to send and receive messages to and from the quagent module. If your quagent

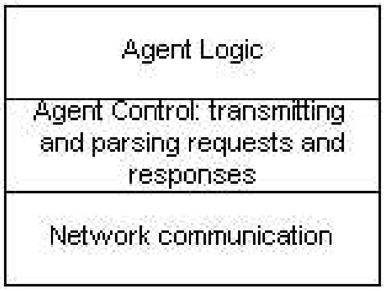

Agent Logic
Agent Control: transmitting and parsing requests and responses
Network communication

Figure 18: Quagent communications layers.

has more sophisticated behavior, you can call an inference engine like Jess in the polymorphic function `Run()`.

`BasicQuagent` cares about sending and receiving messages and doesn't need to know about the underlying communication channels. We make another level of abstraction for network communications. The whole picture looks like Fig. 18

The bottom layer, called `QuagentSocket`, handles the hairy details of interpreting network packages.

The quagent protocol doesn't follow the canonical synchronous server–client convention. As you can see from the above telnet session, asynchronous events occur in an unpredictable way. You'll probably find it handy to use non-blocking communication. The Java Socket provides blocking IO. That is, if you read from the socket and nothing's to be read yet, you will be put waiting until something comes. Say you want to implement `checkEvent`, which just takes a peek at the socket to see if there is an event. If there is, you process the event. If there isn't, you want to continue to issue commands or queries. With blocking communication, you'll be put on hold at check-Event and can't continue with the commands and queries. Non-blocking IO comes in handy here. Non-blocking IO was introduced in JDK1.4 in the java.nio.* package. Client-side non-blocking IO is quite straightforward. We implemented some simple functions, such as read and write strings, in `QuagentSocket.java` using non-blocking socket. Note that, you don't have to use non-blocking IO. The good old Java `Socket` can achieve the same thing, probably with some form of multi-threading.

You might want to insulate yourself from the details of dealing with messages by providing objects and methods that encapsulate the parsing details. We can define a `Request` class and a `Response` class to represent messages to and from the quagent module. Commands (`DO THINGS`) and queries (`ASK THINGS`) are concrete subclasses of `Request`. Different responses are derived from `Response`, too. `BasicQuagent` only takes care of sending requests and receiving responses. The polymorphic functions in the specific requests and responses do the real job. E.g.,

```
public class BasicQuagent {
        ...
        QuagentSocket socket;

        public Request send(Request request) {
```

```
        socket.send(request.toString());
        return request;

    }
}
```

Each command or query composes protocol-abiding messages in their own `toString()` functions. `Send()` returns the request, so you can write more succinct expression like the following

```
Response radius\_response = getResponse(qc.send(radius\_query));
```

B.4 The Sample Code

The sample code is in the `examples` directory of our source distribution. There may be a `make` file to compile and run the sample, or you can simply

```
prompt: javac DrunkenMaster.java
prompt: java DrunkenMaster
```

It tries to connect to a quagent module running at local machine. If you want to connect to a remote machine, change `"localhost"` to the host name of that machine in `DrunkenMaster.java`. We didn't implement all the requests and responses, but the file and class names should make clear what they are.

References

[1] Artificial intelligence course assignments and student projects, September 2004. http://www.cs.rochester.edu/ u/brown/242/assts/assts.html.

[2] Artificial intelligence course main page, September 2004. http://www.cs.rochester.edu/ u/brown/242/242_home.html.

[3] Conversational interaction and spoken dialog research group, September 2004. http://www.cs.rochester.edu/research/cisd.

[4] Doom as a tool for system administration, 2004. http://www.cs.unm.edu/∼lchao/flake/doom/.

[5] The foundation for intelligent physical agents, 2004. http://www.fipa.org.

[6] id software downloads, 2004. http://www.idsoftware.com/business/techdownloads/.

[7] Jess programming system, 2004. http://herzberg.ca.sandia.gov/jess/.

[8] John laird's artificial intelligence and computer games research, 2004. http://ai.eecs.umich.edu/people/ laird/gamesresearch.html.

[9] Machinima.com, 2004. http://machinima.com.

[10] Modeling human prefrontal cortex in quake iii arena, 2004. http://www.vuse.vanderbilt.edu/ ∼noelledc/resources/VE/IDED/home.htm.

[11] Quake army knife, 2004. http://dynamic.gamespy.com/∼quark/.

[12] Quake level editor, 2004. http://www.planetquake.com/worldcraft/index2.shtm.

[13] U. pennsylvania grasp lab, September 2004. http://www.cis.upenn.edu/∼stepligh/overview.html.

[14] U. rochester quagent main page, Sept. 2004. http://www.cs.rochester.edu/ research/quagents/.

[15] J. Allen and G. Ferguson. Human-machine collaborative planning. In *Proceedings 4rd Int. NASA Workshop on Planning and Scheduling for Space*, October 2002.

[16] J. F. Allen. *Natural Language Understanding*. Benjamin Cummings Publishing Co., Menlo Park, 1987.

[17] James Allen, Nate Blaylock, and George Ferguson. A problem-solving model for collaborative agents. In *First International Joint Conference on Autonomous Agents and Multiagent System*, July 2002.

[18] James Allen, Nate Blaylock, and George Ferguson. Synchronization in an asynchronous agent-based architecture for dialogue systems. In *Proceedings of the 3rd SIGdial Workshop on Discourse and Dialog*, July 2002.

[19] James Allen and George Ferguson. Human-machine collaborative planning. In *Proceedings of the 3rd Int. NASA Workshop on planning and scheduling for space*, October 2002.

[20] Mark Armstrong. Optimal multi-object auctions. *Review of Economic Studies*, 67:455–481, 2000.

[21] M. Asada, S. Noda, S. Tawaratumida, and K. Hosoda. Purposive behavior acquisition for a real robot by vision-based reinforcement learning. *Machine Learning*, pages 279–303, 1996.

[22] Tucker Balch and Ronald Arkin. Behavior-based formation control for multi-robot teams. T. Balch and R. Arkin, Behavior-based Formation Control for Multi-robot Teams. Submitted for publication (1997), (citeseer.ist.psu.edu/balch99behaviorbased.html).

[23] P. Barnum, G. Ferguson, and C. Brown. Team coordination with multiple mobile agents. Technical Report URCS TR Draft, Dept. CS, U. Rochester, September 2004.

[24] Nate Blaylock, James Allen, and George Ferguson. *Managing communicative intentions with collaborative problem solving*. Kluwer, 2002.

[25] R. Brooks. A layered intelligent control system for a mobile robot. *IEEE T-Robotics and Automation*, pages 14–23, April 1986.

[26] R.A. Brooks and J.H. Connell. Asynchronous distributed control system for a mobile robot. In *Proc. of SPIE*, volume 727, pages 77–84, 1986.

[27] Rodney Brooks. Intelligence without reason. In *Proceedings of the Int. Joint Conf. on AI*, pages 569–595, 1991.

[28] Rodney Brooks. Intelligence without representation. *Artificial Intelligence*, (47):139–159, 1991.

[29] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2, No 1:14–23, 1986.

[30] Rodney A. Brooks. A robot that walks; emergent behavior from a carefully evolved network. *Neural Computation*, 1(2):253–262, 1989.

[31] M. Buckland. *AI techniques for game playing*. Premier Press, Cincinnati, OH, 2002.

[32] R. L. Carceroni, C. Harman, C. Eveland, and C. M. Brown. *Real–Time Pose Estimation for Convoying Applications*, pages 230–243. The Confluence of Vision and Control, LNCIS Series No. 237. Springer–Verlag, 1998.

[33] credited to Ohio State Student. Untitled: Quake ii calling trees, August 2003.

[34] Randall Davis and Reid Smith. Negotiation as a metaphor for distributed problem solving. *Artificial Intelligence*, 20(1):63–109, January 1983.

[35] M. Bernardine Dias and Anthony Stentz. Traderbots: A market-based approach for resource, role, and task allocation in multirobot coordination. Technical Report CMU-RI-TR-03-19, CMU Robotics Institute, August 2003.

[36] Michael R. Genesereth and Steven P. Ketchpel. Software agents. *Commun. ACM*, 37(7):48–ff., 1994.

[37] Brian P. Gerkey and Maja J. Mataric. Sold!: Auction methods for multi-robot coordination. *IEEE Transactions on Robotics and Automation Special issue on multi-robot systems*, 18(5):758–768, October 2002.

[38] Robert S. Gray. Transportable agents. Gray, Robert. Transportable Agents. Ph.D. Thesis Proposal. Dartmouth College, NH. (1995), url = citeseer.ist.psu.edu/gray95transportable.html.

[39] S. Holmes. *Focus on Mod Programming for Quake III Arena*. Premier Press, Cincinnati, OH, 2002.

[40] Bo Hu and Christopher Brown. Interactive indoor scene reconstruction from image mosaics using cuboid structure. In *IEEE Workshop on Motion and Video Computing, vol 2*, 2002.

[41] Marcus J. Huber and Edmund H. Durfee. On Acting Together: Without Communication. In *Spring Symposium Working Notes on Representing Mental States and Mechanisms*, pages 60–71. American Association for Artificial Intelligence, Stanford, California, 1995.

[42] Luke Hunsberger and Barbara Grosz. A combinatorial auction for collaborative planning. In *Proceedings of the Fourth International Conference on Multi-Agent Systems (ICMAS-2000)*.

[43] Nicholas R. Jennings. Controlling cooperative problem solving in industrial multi-agent systems using joint intentions. *Artificial Intelligence*, 75(2):195–240, 1995.

[44] Olle Kallander. Quake2 game dll documentation.

[45] T. Kollar, J. Schmid, E. Meisner, M. Elsner, D. Calarese, C. Purav, C. Brown, J. Turner, D. Peramunage, G. Altekar, and V. Sweetser. Mabel: Extending human interaction and robot rescue designs. Dspace: http://hdl.handle.net/1802/806.

[46] T. Kollar, J. Schmid, E. Meisner, M. Elsner, D. Calarese, C. Purav, C. Brown, J. Turner, D. Peramunage, G. Altekar, and V. Sweetser. Mabel: Extending human interaction and robot rescue designs, Spring 2004. Dspace: http://hdl.handle.net/1802/805.

[47] J. E. Laird. It knows what you're gong to do: adding anticipation to a quakebot. Technical Report AAAI Tech Rpt SS-00-02, AIII Spring Symposium Series: AI and Interactive Entertainment, March 2000.

[48] J. E. Laird and J. C. Duchi. Creating human-like synthetic characters with multiple skill levels: a case study using the soar quakebot. Technical report, AIII Spring Symposium Series: AI and Interactive Entertainment, March 2001.

[49] J.E. Laird and M. van Lent. Developing an artificial intelligence game engine. In *Proceedings: Game Developer's Conference*, pages 577–588, San Jose, 1999.

[50] Anton Likhodedov and Tuomas Sandholm. Methods for boosting revenue in combinatorial auctions. In *Proceedings on the National Conference on Artificial Intelligence (AAAI)*, 2004.

[51] Jiebo Luo, Matthew Boutell, Robert T. Gray, and Christopher Brown. Using image transform-based bootstrapping to improve scene classification. In *2004 International Symposium on Electronic Imaging*, San Jose, CA, 2004.

[52] (Ed.) Martin Gardner. *Mathematical Puzzles of Sam Loyd*. Dover Publications, Inc, NYC, NY, 1959.

[53] R. Nelson and A. Selinger. A cubist approach to object recognition. In *International Conference on Computer Vision (ICCV98)*, pages 614–621, Bombay, India, January 1998.

[54] Randal Nelson and Isaac Green. Tracking objects using recognition. In *International Conference on Pattern Recognition (ICPR02), vol.2*, pages 1025–1039, Quebec City, Quebec, August 2002.

[55] Anthony Sang-Bum Park. *A service-based agent system supporting mobile computing*. PhD thesis, Heinisch-Westfalischen Technischen Hochschule, 2004. Doctoral thesis.

[56] Ramprasad Polana and Randal C. Nelson. Detection and recognition of periodic, non-rigid mot ion. *International Journal of Computer Vision*, 23(3):261–282, June/July 1997.

[57] Craig W. Reynolds. Flocks, herds, and schools: A distributed behavioral model. *Computer Graphics*, 21(4):25–34, 1987.

[58] S.J. Russell and P. Norvig. *Artificial Intelligence A Modern Approach (2nd Ed)*. Prentice Hall, New Jersey, 2003.

[59] J. Schmid, T. Kollar, E. Meisner, V. Sweetser, D . Feil-Seifer, C. Brown, et al. Mabel: Building a robot designed for human interaction. In William D Smart, editor, *AAAI Workshop Technical Report: AAAI Mobile Robot Competition and Exhibition*, number WS-02-18, pages 24–32, August 2002. In DSpace: http://hdl.handle.net/1802/804.

[60] Andrea Selinger and Randal C. Nelson. A perceptual grouping hierarchy for appearance-based 3d object recognition. *Computer Vision and Image Understanding*, 76(1):83–92, October 1999.

[61] Andrea Selinger and Randal C. Nelson. Appearance-based object recognition using multiple view s. In *Computer Vision and Pattern Recognition (CVPR01), Volum e 1*, pages 905–911, Kauai, Hawaii, December 2001.

[62] Reid G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on Computers*, C-29(12):1104–1113, December 1980.

[63] J. Spletzer, A.K. Das, R. Fierro, C.J. Taylor, V. Kumar, and J.P. Ostrowski. Cooperative localization and control for multi-robot manipulation. In *Proceedings IROS 2001*, December 2001.

[64] N. Sprague. *Learning to coordinate visual behaviors*. PhD thesis, U. Rochester Computer Science Department, August 2004.

[65] N. Sprague and D. Ballard. A visual control architecture for a virtual humanoid. In *IEEE-RAS Int. Conf. on Humanoid Robots*, November 2001.

[66] Luc Steels. Cooperation between distributed agents through self-organization. *Decentralized A.I.*, pages 175–196, 1990.

[67] Milind Tambe. Towards flexible teamwork. *Journal of Artificial Intelligence Research*, 7:83–124, 1997.

[68] D. Terzopolous and F. Rabie. Animat vision: Active vision in artificial animals. *VIDERE*, 1(1), Fall 1997. Electronic Journal: http://www-mitpress.mit.edu/e-journals/Videre/.

[69] Demetri Terzopoulos. Modeling living systems for computer vision. In *Int. Joint Conf. on AI*, pages 1003–1013, 1995. citeseer.ist.psu.edu/terzopoulos95modeling.html.

[70] Xiaoyuan Tu and Demetri Terzopoulos. Artificial fishes: Physics, locomotion, perception, behavior. *Computer Graphics*, 28(Annual Conference Series):43–50, 1994.

[71] Various. Various. In *Modeling Other Agents from Observations*, July 2004. (Columbia U. Workshop at Int. Joint Conf. on Autonomous Agents and Multi-agent systems.

[72] M. Van Wie. Establishing joint goals through observation. In Carnegie Mellon University David P. Casasent, editor, *Intelligent Robots and Computer Vision XVI:Algorithms, Techniques, Active Vision, and Materials Handling*, Pittsburgh, PA, USA, October 1997. The International Society for Optical Engineering.

[73] Michael Van Wie. *Role Selection in Teams of Non-communicating Agents*. PhD thesis, U. of Rochester Dept. of Computer Science, 2000. In DSpace: http://hdl.handle.net/1802/803.

[74] R. Wisniewski. *Achieving high performance in parallel applications via kernel-application interaction*. PhD thesis, U. Rochester Computer Science Department, August 1994.

[75] Robert W. Wisniewski and Christopher M. Brown. Ephor, a run-time environment for parallel intellige nt applications. In *Proceedings of The IEEE Workshop on Parallel and Distributed Real-Time Systems*, pages 51–60, Newport Beach, California, April 13-15, 1993.

[76] Robert W. Wisniewski and Christopher M. Brown. An argument for a runtime layer in sparta design. In *Proceedings of The IEEE Workshop on Real-Time Opera ting Systems and Software*, pages 91–95, Seattle, Washington, May 18-19 1994.

[77] Robert Michael Zlot, Anthony Stentz, M. Bernardine Dias, and Scott Thayer. Multi-robot exploration controlled by a market economy. In *IEEE International Conference on Robotics and Automation*, May 2002.

[78] Gilad Zlotkin and Jeffrey S. Rosenschein. Mechanisms for automated negotiation in state oriented domains. *Journal of Artificial Intelligence Research*, 5:163–238, 1996.