QUAGENTS

A Game Platform for Intelligent Agents

Berardi Francesco
Matricola 0000230775

INTRODUZIONE

L'infrastruttura *Quagents* è stata sviluppata nel 2004 presso l'università di Rochester, negli Stati Uniti. Rilasciata nel novembre dello stesso anno e liberamente distribuibile, l'infrastruttura si propone di implementare un'interfaccia flessibile tra le funzionalità tipiche di un *game engine*, come il rendering grafico delle mappe e la gestione delle collisioni tra oggetti e giocatori, e gli *agenti* che si troveranno a coesistere nell'ambiente virtuale costituito dalle mappe di gioco.
Obiettivo dichiarato del progetto *Quagents* è infatti il trasformare i videogame interattivi tridimensionali da semplici passatempi in utili strumenti per la ricerca e l'insegnamento nel campo dell'intelligenza artificiale.
Nell'ambito dell'attività svolta ci si è quindi proposti lo studio di questa interessante infrastruttura, la sua compilazione in ambiente *GNU/Linux* per renderla pienamente operativa ed infine la realizzazione di un esempio di utilizzo, un'implementazione semplificata del noto gioco *Guardie e Ladri*, in cui un agente, che impersona il ladro, tenta di impossessarsi di un oggetto obiettivo, collocato in un punto a lui sconosciuto della mappa di gioco e difeso da un secondo agente, il guardiano. Una volta carpito l'oggetto, il ladro tenta di fuggire dal livello, tornando al punto di partenza; la simulazione termina con la vittoria del ladro se questi riesce a raggiungere il punto di partenza con l'oggetto obiettivo, con la vittoria del guardiano se quest'ultimo riesce invece a catturare il ladro.

L'INFRASTRUTTURA QUAGENTS

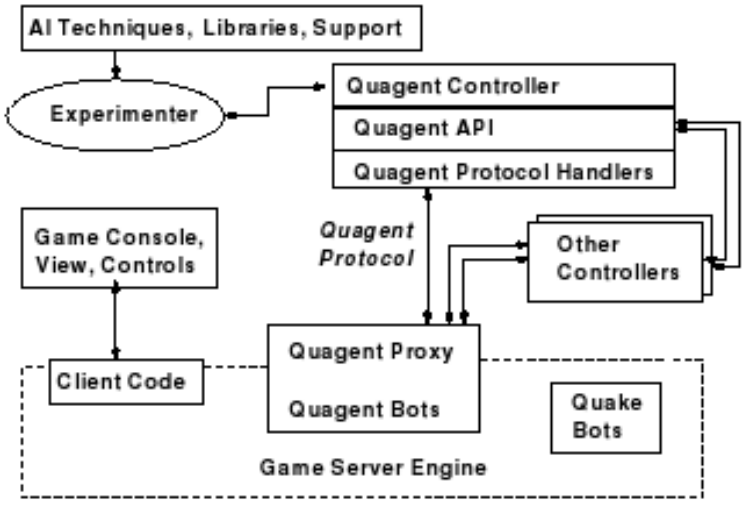
Il sistema *Quagents* è un'infrastruttura per la gestione dell'interazione tra agenti software ed il mondo virtuale in cui questi coesistono che sfrutta le capacità grafiche e di gestione della fisica di *game engine* commerciali sottostanti per fornire ai ricercatori un ambiente grafico di simulazione del tutto simile a quello degli attuali *FPS (First Person Shooters)*.
In realtà, nella sua implementazione attuale, il sistema non è indipendente dal *game engine* sottostante, ma si appoggia ad una versione modificata del videogame *Quake II*, detta *UR-Quake*; la scelta del motore grafico è ricaduta su *Quake II* semplicemente perchè si tratta di un software attualmente *free*, il cui codice sorgente è stato recentemente rilasciato da *ID Software* sotto licenza *GPL* ed è quindi liberamente studiabile e modificabile.
Una volta avviati il motore grafico e l'infrastruttura, quest'ultima apre dei *socket* di ascolto su alcune porte della macchina ospite e si pone in attesa delle richieste degli agenti software; si instaura quindi un dialogo tra agenti ed infrastruttura basato su un protocollo di comunicazione testuale su porte *TCP*, che permette quindi di fatto di realizzare gli agenti in qualsivoglia linguaggio di programmazione, purché siano supportati la gestione dei *socket* e la manipolazione delle stringhe di testo.

UR-QUAKE

UR-Quake è una versione modificata del videogame *Quake II*, sviluppato dalla *ID Software*. Dinamicamente, *Quake II* può essere diviso in una sezione *server* ed una sezione *client*: il client ha il compito di gestire il rendering e la visualizzazione grafica e l'interazione da parte dell'utente attraverso tastiera, mouse, joystick o gamepad, mentre il server, una volta creata un'*immagine* dello stato attuale del mondo virtuale, si preoccupa di mantenerla aggiornata nel tempo e di comunicare eventuali variazioni al client, per permettergli di aggiornare a sua volta la visualizzazione grafica.
In base alla modalità di gioco scelta (*single player* o *multiplayer*) si possono avere un server ed un client coesistenti sulla stessa macchina, oppure un server ed un numero imprecisato di client distribuiti su più macchine ed interagenti tra loro via *LAN* o *Internet*.
Staticamente, *Quake II* consiste di un file eseguibile e due *DLL* (*Dynamical Link Library*): l'eseguibile (*quake2*) contiene al suo interno sia il codice del server che quello del client; la libreria *ref_softx.so* fornisce le funzioni per la gestione della grafica, mentre la libreria *gamei386.so* contiene le funzioni relative agli oggetti, ai mostri ed ai giocatori presenti nel mondo virtuale, comprese le funzionalità per la creazione di *BOT* personalizzati, a cui si appoggia *Quagents*.
Tutti i dati statici del videogame, come mappe, texture degli oggetti e dei personaggi, posizioni iniziali dei mostri, sono infine contenuti in file compresso con estensione *pak*, che purtroppo non è liberamente distribuibile assieme al codice sorgente.
E' possibile dividere il mondo virtuale di *Quake* in oggetti *statici*, come muri, pavimenti, o il cielo, ed entità *mobili*, come mostri, personaggi, armi. Il server mantiene una rappresentazione del mondo attraverso una struttura dati, contenente informazioni relative a tutti gli oggetti attualmente presenti nel mondo; si tratta di una struttura aperta, in quanto in ogni momento è possibile caricare nuovi oggetti esterni al *game engine*, detti *shared objects*, ed aggiungerli alla struttura rendendoli di fatto parte integrante del mondo virtuale. Questo permette l'utilizzo del motore di gioco all'intero del progetto *Quagents*: è sufficiente racchiudere l'infrastruttura all'interno di un oggetto che verrà integrato nel mondo di *Quake* a tempo di esecuzione.
Tutte le entità nel gioco utilizzano una funzione *think* per eseguire autonomamente i calcoli relativi alla gestione del loro comportamento. Il server fornisce ad ogni entità un lasso di tempo per permettergli di eseguire le istruzioni della sua *think*; allo scadere del tempo il controllo passa all'entità successiva e così via in un *loop* infinito di esecuzione; questo permette al server di essere eseguito come un semplice programma *singlethread*. Le entità hanno comunque la possibilità di comunicare al server la necessità di disporre di un tempo di esecuzione più lungo del normale o, al contrario, di venire saltate al successivo *loop* in quanto non necessitano di effettuare calcoli.
Parallelamente alla gestione del comportamento delle entità, il *game engine* si preoccupa dell'aggiornamento del loro aspetto visivo: ogni 100ms viene richiamata una funzione di *RunFrame* che a sua volta richiama un *tree* di funzioni per l'animazione di tutte le entità. Inoltre, ogni 50ms il motore richiama una funzione chiamata *ClientThink*, che legge dal client le azioni intraprese dal giocatore ed aggiorna la posizione relativa delle entità nel mondo in accordo agli spostamenti effettuati. Infine vengono effettuati i calcoli relativi alla fisica del mondo (in particolare vengono determinate le collisioni tra oggetti e personaggi) e finalmente il server invia ai client interessati gli aggiornamenti relativi alle posizioni ed alle orientazioni delle entità del mondo.
La modifica principale introdotta da *UR-Quake* al *game engine* standard di *Quake II*

consiste nella possibilità di caricare *BOT* con maggiori capacità di gestione della rete rispetto al normale, per permettere di fatto di avviare l'infrastruttura *Quagents* come un *BOT shared object* all'interno del mondo virtuale, da dove può godere di una visione globale del mondo e contemporaneamente agire sulla sua evoluzione.

L'ARCHITETTURA QUAGENTS



QUAGENTS ED I BOTS

Come si è visto, in *Quake II* i *BOT* dispongono di una sorta di intelligenza innata ma limitata, che ne permette la gestione autonoma del comportamento. Nel videogame reale, questa intelligenza risiede all'interno della funzione *think*, richiamata dal server ad intervalli regolari, ed è modificabile dall'utente attraverso la regolazione del livello di difficoltà del gioco.
Nell'implementazione di *Quagents* tutta l'intelligenza si trova invece all'esterno del gioco, in programmi scritti dall'utente in un qualunque linguaggio di programmazione (Java, Perl, Lisp, Prolog, C, C++, ...); i *BOT* sono pilotabili attraverso un set finito di comandi a cui essi possono rispondere per notificare dati richiesti dal programma o eventi occorsi nel mondo virtuale.
Come già accennato, la comunicazione tra agenti e *BOT* all'interno del mondo avviene attraverso un protocollo basato su stringhe di testo contenenti parole chiave che identificano le richieste e le risposte e parametri ed esse collegati. Grazie al protocollo *TCP* utilizzato come base per lo scambio di informazioni, è possibile caricare contemporaneamente nel mondo virtuale un numero virtualmente infinito di *BOT*: ognuno di essi disporrà della sua *linea privata* di comunicazione e non potrà influenzare il comportamento degli altri.

LE ENTITA' DEL MONDO VIRTUALE

Nell'infrastruttura *Quagents*, le entità statiche e dinamiche di *Quake II* mantengono il loro significato originario, eventualmente arricchito da nuove funzionalità aggiornabili al momento dell'esecuzione.

- Tempo di esecuzione: il tempo nell'architettura *Quagents* è suddiviso in *tick* della durata di 1/10 di secondo; tutte le azioni che avvengono all'interno dello stesso *tick* si considerano contemporanee.
- Client: il motore di gioco continua a gestire il personaggio principale, che viene però utilizzato come una sorta di *telecamera mobile* gestita dall'utente. Il personaggio viene controllato nel modo consueto dall'utente, che può quindi muoversi liberamente per la mappa di gioco, interagire con i *BOT*, utilizzare le proprie armi, e così via. Unica limitazione non può raccogliere gli oggetti presenti nella mappa, per evitare che modifichi intenzionalmente o meno il mondo virtuale visto dagli agenti.
- Muri e finestre: continuano a mantenere la loro funzione di entità non attraversabili dai *BOT*.
- Oggetti: *box, gold, tofu, battery, data, kryptonite, head*. Questi oggetti possono essere allocati in fase di configurazione o a runtime in punti specifici della mappa di gioco, ognuno di essi ha effetti particolari sui *BOT* al momento della loro raccolta.
- *BOT* multipli: sfruttando le caratteristiche *multiplayer* del *game engine* è possibile caricare e controllare contemporaneamente più *BOT* di osservazione attraverso i diversi *controller* supportati da *Quake II*.

STATO INTERNO DEI BOT

I *BOT* dispongono di uno stato interno contenente informazioni riguardo alla loro posizione e stato fisico, influenzato dagli eventi che si verificano nel mondo virtuale e dagli oggetti che essi raccolgono. Le variabili monitorate sono:

- *Position*, nella forma (X,Y,Z).
- *Orientation,* nella forma (*Roll, Pitch, Yaw*); il valore più importante dei tre è *yaw*, la direzione corrente del *BOT*, in quanto la maggior parte dei comandi e delle risposte fornite dai *BOT* contengono delle coordinate ottenute considerando la posizione corrente come origine e la direzione corrente come angolo di partenza.
- *Velocity.*
- *Inventory*, la lista degli oggetti posseduti dal *BOT*.
- *Health.*
- *Wealth.*
- *Wisdom.*
- *Energy.*
- *Age.*

LEGGI FISICHE E BIOLOGICHE

Tutte le leggi fisiche implementate nel *game engine* di *Quake II* continuano ad essere utilizzate nell'infrastruttura *Quagents*; si tratta in generale delle leggi che governano le collisioni tra i *BOT* e le entità presenti nel mondo: oggetti, altri *BOT*, muri, finestre e pavimenti.
Vengono inoltre implementate numerose altre leggi, in massima parte rivolte alle variazioni nel tempo delle caratteristiche biologiche dei *BOT:*

- Un *BOT* nasce nel mondo virtuale al momento della connessione dell'agente software con l'infrastruttura; la posizione iniziale del *BOT* è configurabile attraverso un file di configurazione.
- La disconnessione dell'agente comporta la morte istantanea del *BOT*, il cui corpo rimane visibile nel mondo virtuale.
- Tutti gli oggetti a parte le casse, i muri e le finestre sono attraversabili dai *BOT*.
- Tutti gli oggetti possono essere raccolti.
- Raccogliere (abbandonare) un oggetto *Tofu* porta all'incremento (decremento) del parametro *Health*.
- Raccogliere (abbandonare) un oggetto *Gold* porta all'incremento (decremento) del parametro *Wealth*.
- Raccogliere (abbandonare) un oggetto *Data* porta all'incremento (decremento) del parametro *Wisdom*.
- Raccogliere (abbandonare) un oggetto *Battery* porta all'incremento (decremento) del parametro *Energy*.
- Raccogliere un oggetto *Kryptonite* porta al decremento del parametro *Energy,* come se il *BOT* si trovasse nelle vicinanze dell'oggetto.
- L'incremento del parametro *Wisdom* comporta un aumento percentuale dell'*abilità* del *BOT*, in particolare il massimo raggio di visuale nel comando *RADIUS* ed il massimo numero di raggi di vista calcolati dal comando *RAYS*.
- L'età del *BOT*, registrata dal parametro *Age*, aumenta ad ogni *tick* del tempo di esecuzione; ogni *BOT* dispone di una propria età massima, *lifetime*, raggiunta la quale muore di vecchiaia.
- Il parametro *Energy* decresce di un piccolo quantitativo ad ogni *tick*; quando l'energia scende a zero il *BOT* diviene incapace di muoversi: tutte le richieste di movimento verranno accettate, ma il *BOT* si muoverà sempre di 0 unità. Tutte le altre capacità del *BOT* rimangono attive, compresa la possibilità di raccogliere oggetti. Altri *BOT* possono quindi fornirgli batterie per aumentare di nuovo la sua energia.
- Il parametro *Health* decresce quando il *BOT* subisce dei colpi di arma da fuoco o se cade da altezze notevoli; se il parametro scende a zero il *BOT* muore.
- Se il *BOT* si viene a trovare nelle vicinanze di un oggetto *Kryptonite*, la sua energia inizia a calare di un quantitativo dipendente dalla distanza dell'oggetto ad ogni *tick*, sino a quando il *BOT* non si sposta al di fuori del raggio d'azione dell'oggetto.

FILES DI CONFIGURAZIONE

L'infrastruttura *Quagents* prevede due diverse tipologie di files di configurazione.
Il primo tipo di files è rivolto alla configurazione iniziale del mondo virtuale ed al suo popolamento con i vari tipi di oggetti: uno ed un solo file di questo tipo viene sempre caricato all'avvio dell'infrastruttura; qualora il file non venisse trovato in una serie di percorsi predefiniti, contenente la *home* dell'utente e la directory in cui risiedono gli agenti, verrà letto un file di *default.*
Il file di configurazione deve necessariamente chiamarsi *config.dat* ed ha un formato molto semplice: per ogni oggetto che si vuole caricare deve essere aggiunta al file una linea del tipo

```
[nome oggetto] [x] [y] [z]
```

dove [nome oggetto] è una parola chiave tra BOX, TOFU, BATTERY, GOLD, DATA, KRYPTONITE, e [x] [y] [z] sono stringhe interpretate come le coordinate della posizione dell'oggetto. E' importante sottolineare che l'*engine* non effettua alcun controllo sulla validità delle coordinate, questo significa che è necessario conoscere a priori la mappa di gioco utilizzata e scrivere nel file di configurazione dei valori coerenti per evitare di posizionare oggetti all'esterno della mappa o all'interno di muri.
Il caricamento di oggetti nella mappa di gioco può anche essere effettuato a *runtime* attraverso la console di *Quake II*.

Il secondo tipo di files di configurazione riguarda invece le caratteristiche dei *BOT*. Ogni volta che un *BOT* viene caricato, dovrebbe essere fornito all'infrastruttura anche un adeguato file di configurazione, contenente delle righe di testo nel formato

```
[variabile] [valore]
```

dove [variabile] è una parola chiave tra
- LIFETIME
- INITIALWISDOM
- INITIALENERGY
- INITIALWEALTH
- INITIALHEALTH
- ENERGYLOWTHRESHOLD
- AGEHIGHTHRESHOLD

e [valore] è una stringa interpretata come il valore iniziale da assegnare al parametro specificato.
Tutti i parametri possono essere o non essere presenti nel file, in quest'ultimo caso alla variabile di stato corrispondente viene assegnato un valore di *default*, e possono essere specificati in qualunque ordine.
Molto importante è anche il parametro che specifica la posizione iniziale del *BOT*, espresso nella forma:

```
INITIALLOCATION [x] [y] [z]
```

Qualora al momento del caricamento di un *BOT* non dovesse venire specificato alcun file di

configurazione, questo apparirà nella stanza di *default* della mappa (normalmente il punto di partenza del livello) in una posizione e con parametri di *default.*

Segue un esempio di un possibile file di configurazione globale *config.dat*:

```
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
```

COMUNICAZIONE ED INTERAZIONE TRA AGENTI

Allo stato di sviluppo attuale dell'infrastruttura *Quagents*, la comunicazione tra agenti risulta particolarmente semplice se tutti gli agenti interessati sono *thread* avviati dallo stesso programma, mentre potrebbe essere un poco più difficoltosa e soprattutto non uniforme in caso contrario.
Per mantenere infatti l'astrazione degli agenti da quello che è il *game engine*, non vengono utilizzate tutte le facilitazioni per la comunicazione tra utenti del videogame messe a disposizione dal programma, come la *chat interna*: in *Quagents* la comunicazione deve avvenire a livello di agenti, non di *BOT*.
Il sistema di comunicazione tra agenti attualmente implementato prevede che un particolare agente possa inviare un *messaggio* interpretabile dagli altri agenti in tre diverse modalità:

- Broadcast: il messaggio viene inviato a tutti gli agenti attualmente connessi.
- Singolo destinatario: il messaggio viene inviato ad un destinatario specifico.
- Destinatari multipli: il messaggio viene inviato a tutti gli agenti i cui *BOT* si trovano entro una determinata distanza dal mittente.

L'esistenza di diverse modalità di invio dei messaggi permette di implementare forme di cooperazione tra agenti, estremamente utili in tutte quelle simulazioni in cui più squadre di agenti devono fronteggiarsi per raggiungere degli obiettivi: l'invio di messaggi entro una certa distanza permetterebbe di comunicare con i compagni di squadra senza farsi udire dai nemici, così come l'invio a destinatari specifici rende possibili comunicazioni private.
La non uniformità delle comunicazioni deriva però dal fatto che, mentre sono implementati i comandi per l'invio e la ricezione dei messaggi, il *contenuto* dei messaggi stessi non è sottoposto ad alcuna regola di validazione, ma viene completamente deciso dall'agente che effettua l'invio.
E' quindi suo compito assicurarsi che tutti gli agenti a cui il messaggio è indirizzato siano in grado di comprenderne correttamente il contenuto informativo.

PROTOCOLLO DI COMUNICAZIONE

I *BOT* vengono controllati e comunicano tra loro sfruttando un particolare protocollo di comunicazione, denominato *Quagent Protocol*; è importante sottolineare come in ogni caso il protocollo non sostituisce l'interazione con il *game engine* attraverso la console di gioco integrata in *Quake II*: mentre quest'ultima permette di inviare comandi specifici al videogame e di ricevere le relative risposte (ad esempio la lettura della posizione corrente del *BOT*, l'attivazione di una specifica arma, l'assegnazione di un tasto ad una particolare azione, l'aumento del volume degli effetti audio), il protocollo di comunicazione permette di realizzare un collegamento bidirezionale tra il *controller* ed i *BOT*.
Il software di controllo può inviare al *BOT comandi* o *richieste*; questo può quindi semplicemente confermare la ricezione dei comandi, inviare messaggi di errore, o spedire al mittente della richiesta dati di varia natura come risposta. I *BOT* inoltre posso volontariamente decidere di notificare al software di controllo il verificarsi di particolari eventi.
La forma generale dei messaggi utilizzati dal protocollo è la seguente.

Dal *controller* verso il *BOT*:

- DO [Comando] [Parametri]
- ASK [Richiesta] [Parametri]

Dal *BOT* verso il *controller*:

- OK ([Echo]) (Conferma di ricezione di un comando, viene inviato al *controller* anche l'eco del comando stesso)
- ERR ([Echo]) [Descrizione dell'errore] (Impossibile iniziare l'esecuzione del comando)
- TELL [Evento] [Parametri] (Informazioni volontariamente inviate dal *BOT*, messaggio asincrono)

Nelle righe precedenti e nel seguito, le stringhe in caratteri maiuscoli indicano parole chiave dei messaggi, le stringhe tra parentesi quadre sono segnaposto per gli argomenti dei messaggi, eventualmente interpretati come valori numerici, interi o reali.
La risposta OK ([Echo]) o ERR ([Echo]) [Descrizione dell'errore] è detta *risposta standard*, con la quale il *BOT* nel primo caso segnala la sua disponibilità a tentare di eseguire il comando indicato, nel secondo caso rifiuta l'esecuzione e ne illustra il motivo; la stringa *Echo*, sempre racchiusa tra parentesi tonde, riporta testualmente il comando inviato precedentemente al *BOT* a cui la risposta si riferisce, per evitare confusione in caso di ricezioni multiple.

COMANDI DI AZIONE

- WALKBY [distanza] (ad esempio WALKBY 20.0)
  Descrizione: invita il *BOT* ad iniziare a camminare nella direzione corrente; nel migliore dei casi, il *BOT* inizierà a camminare nella direzione desiderata per fermarsi poi silenziosamente una volta percorsa la distanza indicata. Se il *BOT* è costretto a fermarsi prima di aver percorso l'intera distanza, ad esempio per la presenza di ostacoli sul suo cammino, notificherà l'evento al *controller*, informandolo anche dell'effettiva distanza percorsa. La trasposizione grafica del *BOT* all'interno di *Quake II* prevede che lo spostamento del personaggio nella mappa di gioco sia accompagnato da un'animazione ciclica del personaggio stesso per simularne il movimento; poiché l'animazione, gestita interamente dal *game engine*, avanza di un fotogramma ad ogni passo del personaggio e può essere interrotta esclusivamente in determinati fotogrammi chiave, l'indicazione della distanza da percorrere può essere solo approssimativamente rispettata: il *BOT* terminerà il suo cammino dopo aver percorso la distanza che meglio approssima quella richiesta compatibilmente con l'avanzare dell'animazione grafica. Normalmente non è quindi buona norma implementare nel software di controllo un calcolo autonomo della posizione attuale del *BOT*, ma è conveniente attendere di volta in volta le comunicazioni del *BOT* stesso.
  Risposta: standard.

- RUNBY [distanza]
  Descrizione: simile a WALKBY sia nel comportamento che nelle risposte fornite, invita il *BOT* a percorrere la distanza indicata correndo anziché camminando.
  Risposta: standard.

- STAND
  Descrizione: comanda al *BOT* di arrestarsi, qualunque fosse il comando di movimento in corso. Alla ricezione del comando il *BOT* interrompe ogni azione che stava compiendo.
  Risposta: standard.

- TURNBY [angolo di rotazione]
  Descrizione: invita il *BOT* a ruotare su se stesso dell'angolo indicato; l'angolo di rotazione è espresso in gradi, se positivo indica una rotazione a sinistra, se negativo a destra. L'esecuzione del comando implica una variazione del parametro di stato *yaw* (orientazione) del *BOT*.
  Risposta: standard.

- PICKUP [nome dell'oggetto]
  Descrizione: invita il *BOT* a raccogliere l'oggetto più vicino a lui il cui tipo coincide con la stringa passata come parametro; se nessun oggetto del tipo indicato si trova nel raggio di raccolta del *BOT* sopraggiunge una condizione di errore che viene segnalata al *controller*.
  [nome dell'oggetto] deve essere una delle seguenti parole chiave: BOX, GOLD, TOFU, BATTERY, DATA, KRYPTONITE, HEAD.
  Risposta: standard.

- DROP [nome dell'oggetto]
  Descrizione: comando opposto a PICKUP, invita il *BOT* a depositare a terra, nella posizione corrente in cui si trova, uno degli oggetti contenuti nel suo inventario il cui tipo corrisponde alla stringa passata come parametro; una condizione notificata al *controller* si verifica se nell'inventario del *BOT* non è presente alcun oggetto del tipo indicato.
  [nome dell'oggetto] deve essere una delle parole chiave elencate per il comando PICKUP. *Risposta:* standard.

COMANDI DI PERCEZIONE

- ASK RADIUS [distanza]
  Descrizione: invita il *BOT* a comunicare quali oggetti sono nel suo raggio di vista ed entro la distanza indicata; naturalmente il *game engine* impone una distanza massima entro cui un oggetto è osservabile.
  Risposta: ERR ([echo]) [descrizione dell'errore] se il comando fallisce, o OK ([echo]) [numero di oggetti osservati] ([nome dell'oggetto] [posizione relativa])*, dove (...)* indica un array di dimensioni non note; per ogni oggetto osservato, l'array viene popolato con il nome dell'oggetto stesso e la sua posizione relativa a partire dalla posizione corrente del *BOT*, nella forma di un vettore tridimensionale di distanze (x,y,z).
  Ad esempio: OK (ASK radius 100.0) 2 GOLD -20.0 30.0 0 Sandhya -320 -100 0.

- ASK RAYS [numero di raggi]
  Descrizione: invita il *BOT* a comunicare quali entità si trovano attorno al *BOT*; contrariamente al comando RADIUS, RAYS segnala anche la presenza delle entità che compongono la mappa di gioco, come muri o finestre, dette *world spawn*. Alla ricezione del comando, il *BOT* distribuisce uniformemente attorno a se il numero di raggi richiesti e verifica la presenza di entità lungo ciascuno dei raggi; aumentare il numero dei raggi porta ad una maggiore accuratezza dell'osservazione, ma allunga anche i tempi di calcolo della risposta.
  Risposta: analoga a quella del comando ASK RADIUS, con l'aggiunta del tipo di oggetto *world spawn* ad indicare elementi della mappa di gioco.
  Ad esempio: OK (ASK RAYS 2) 1 world spawn 315.0 277.1 0.0 2 TOFU 200 100 0.

- ASK PING [numero di raggi]
  Descrizione: versione ad alta risoluzione del comando ASK RAYS, del tutto analogo a questo per quanto riguarda comportamento e risposte fornite.
  Risposta: analoga al comando ASK RAYS.

- CAMERAON
  Descrizione: normalmente la finestra grafica del videogame mostra la visuale di gioco dal punto di vista del personaggio controllato dall'utente; l'esecuzione del comando sposta la telecamera sul *BOT*, permettendo di osservare il mondo di gioco dal suo punto di vista.

Risposta: standard.

- CAMERAOFF
  Descrizione: comando opposto a CAMERAON, riporta la telecamera sul personaggio controllato dall'utente.
  Risposta: standard.

- LOOK
  Descrizione: aggiorna il buffer grafico della visuale attuale del *BOT*; il comando viene normalmente eseguito al termine di CAMERAON, ma può comunque essere richiamato in ogni momento in modo indipendente.
  Risposta: standard.

RICHIESTE E COMUNICAZIONI ASINCRONE

Al di la dei comandi di movimento e di percezione, con cui si invita il *BOT* compiere determinate azioni, spesso risulta molto utile inviare al *BOT* delle richieste per verificare ed eventualmente modificare il suo stato interno. Segue la lista dei comandi preposti allo scopo.

- GetWhere
  Descrizione: richiede al *BOT* di comunicare la sua posizione attuale, comprensiva di orientamento, direzione e velocità correnti.
  Risposta: ERR ([echo]) [descrizione dell'errore] in caso di fallimento, o OK ([echo]) [vettore di stato], dove [vettore di stato] è un vettore contenente la posizione attuale ed i parametri di stato del *BOT* nella forma (world x, world y, world z, roll, pitch, yaw, velocity).

- GetInventory
  Descrizione: richiede al *BOT* di comunicare il contenuto del suo inventario.
  *Risposta:*un vettore contenente la lista degli oggetti posseduti dal *BOT*.

- GetWellbeing
  Descrizione: richiede al *BOT* di comunicare quali azioni sta compiendo.
  Risposta: la descrizione dell'azione corrente che il *BOT* sta intraprendendo.

- SetEnergyLowThreshold [valore]
  Descrizione: imposta al valore specificato la soglia minima di energia al di sotto della quale il *BOT* non è più in grado di compiere azioni; la soglia è comunque selezionabile a *design time* anche attraverso il file di configurazione del *BOT*. Quando l'energia scende al di sotto della soglia, il *BOT* segnala l'evento inviando al *controller* un singolo messaggio di tipo TELL.
  Risposta: standard.

- SetAgeHighhreshold [valore]
  Descrizione: imposta al valore specificato la massima età che il *BOT* può raggiungere prima di morire di vecchiaia; il valore è comunque configurabile anche a

design time attraverso il file di configurazione del *BOT*. Quando l'età del *BOT* supera la soglia, il *BOT* notifica l'evento al *controller* inviandogli un singolo messaggio di tipo TELL.
Risposta: standard.

In risposta a particolari eventi che possono sopraggiungere in modo asincrono nel mondo di gioco, i *BOT* hanno la capacità di inviare volontariamente al *controller* dei messaggi di notifica.
Tali messaggi, detti di tipo TELL, si presentano nella forma:

TELL [condizione]

dove [condizione] è una stringa di testo che descrive l'evento occorso nel mondo.
Le seguenti condizioni sono riconosciute dai *BOT* e portano all'invio di notifiche al *controller*:

- STOPPED
  Descrizione: il *BOT* stava eseguendo un comando di movimento ma è stato costretto ad arrestarsi prima di percorrere la distanza richiesta a causa di ostacoli che si sono trovati sul suo cammino.

- KRYPTONITENEAR
  Descrizione: almeno un oggetto di tipo Kryptonite si trova nell'area considerata *vicina* al *BOT*, dove ha i suoi massimi effetti. Normalmente la notifica segue ad una di tipo KRYPTONITEFAR.

- KRYPTONITEFAR
  Descrizione: almeno un oggetto di tipo Kryptonite si trova nell'area considerata *lontana* dal *BOT*, dove ha un effetto limitato sulla salute del *BOT* stesso. Normalmente, se il *BOT* prosegue nel suo movimento, la notifica è seguita da una di tipo KRYPTONITENEAR.

- KRYPTONITENOT
  Descrizione: almeno un oggetto di tipo Kryptonite, precedentemente rilevato nell'area *lontana* dal *BOT*, è ora uscito dal raggio di effetto sul *BOT* stesso, che non lo considera quindi più un pericolo per la sua salute.

- LOWENERGY
  Descrizione: il livello di energia del *BOT* è sceso al di sotto della soglia minima; il *BOT* tra poco verrà a trovarsi in una condizione di paralisi e non sarà più in grado di eseguire alcun comando di movimento sino a quando l'energia non tornerà al di sopra della soglia.

- NOTLOWENERGY
  Descrizione: il livello di energia del *BOT*, precedentemente al di sotto della soglia minima, è nuovamente tornato ad un livello accettabile; il *BOT* è di nuovo in grado di compiere azioni di movimento.

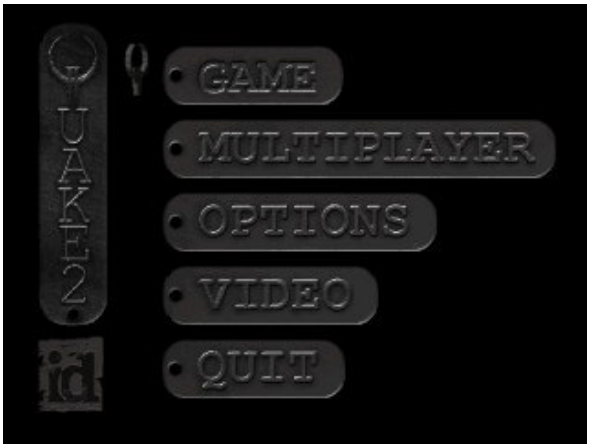
- OLDAGE
  Descrizione: l'età del *BOT* ha superato la soglia massima, la morte per vecchiaia del *BOT* è imminente. Al contrario dell'eccessivo calo dell'energia, condizione da cui il *BOT* è in grado di uscire raccogliendo delle celle energetiche, lo stato di superamento dell'età massima è irreversibile e porta irrimediabilmente ad una notifica di tipo DYING a cui segue la morte del *BOT*.

- DYING: [Parametri di stato] [Inventario] [Azione corrente]
  Descrizione: il *BOT* sta morendo di vecchiaia a causa del superamento dell'età massima; all'invio della notifica segue l'esecuzione dell'animazione grafica della morte del personaggio e la cessazione dell'invio di ogni risposta e notifica al *controller*. Questa notifica segue sempre ad una di tipo OLDAGE.

- STALLING: [Parametri di stato] [Inventario] [Azione corrente]
  Descrizione: il *BOT* è paralizzato poiché il livello energetico è rimasto per troppo tempo al di sotto della soglia minima; in queste condizioni non è più in grado di eseguire alcun comando di movimento sino a quando l'energia non tornerà al di sopra della soglia.
  Questa notifica segue sempre ad una di tipo LOWENERGY.

INTERAZIONE CON UR-QUAKE E QUAGENTS

Dal punto di vista applicativo, l'interazione con gli agenti *Quagents* avviene attraverso lo scambio di messaggi di testo composti esclusivamente da caratteri ASCII. Questo permette di realizzare il software di controllo in qualunque linguaggio si ritenga opportuno, purché siano supportati i *socket* e la gestione delle porte di comunicazione su rete locale o internet; all'estremo si potrebbe anche semplicemente stabilire una connessione con i *BOT* mediante *Telnet* ed utilizzarlo per scambiare i messaggi di controllo.



Per utilizzare *Quagents* è necessario che tutti i software utilizzati siano avviati nell'ordine corretto. In particolare si deve in primo luogo avviare il videogame *Quake II* ed iniziare una nuova partita in modalità *multiplayer*, ad un qualunque livello di difficoltà: questo abilita la possibilità di caricare *BOT* personalizzati nella mappa di gioco. La versione modificata di *Quake II* che andiamo ad utilizzare, *UR-Quake*, procede autonomamente al caricamento del *BOT* principale dell'infrastruttura *Quagents* all'avvio di ogni partita in modalità *multiplayer*.
A questo punto l'infrastruttura è completamente abilitata e la mappa di gioco risulta già esplorabile muovendosi nel modo consueto, sfruttando cioè i tasti W S A D per muoversi ed il mouse per guardarsi attorno, il personaggio giocante; tutte le opzioni del videogame, come i tasti collegati alle varie azioni, le impostazioni grafiche e sonore, sono modificabili dagli appositi *menu*, come nella versione originale del gioco.

E' quindi possibile avviare gli agenti di controllo dei *BOT*, i quali dovranno in un primo momento aprire un *socket* di comunicazione verso l'*host* che ospita il server sulla porta desiderata (di default la 33333), e successivamente monitorare in un ciclo continuo il canale di comunicazione per gestire la ricezione e l'invio dei messaggi da e verso i *BOT*. E' importante sottolineare come, una volta aperto un canale di comunicazione, questo venga permanentemente monitorato dal *server*: qualora la connessione dovesse cadere per un qualunque motivo, il *BOT* controllato verrebbe immediatamente ucciso ed il canale chiuso; una comunicazione interrotta non è quindi ripristinabile.
Non appena *UR-Quake* viene avviato si appropria del controllo del mouse: i movimenti del mouse all'interno della finestra del videogame controllano infatti il personaggio dell'utente; per portare il puntatore all'esterno della finestra è necessario mantenere premuto il tasto *shift*, che rilascia il controllo del mouse al sistema operativo per tutto il tempo in cui il tasto viene mantenuto premuto.
Per tornare poi al controllo del videogame una volta terminate le operazioni all'esterno della finestra è sufficiente cliccare con il mouse all'interno della finestra, prestando attenzione al fatto che questa operazione provoca l'esplosione di un colpo di arma da fuoco da parte del personaggio controllato dall'utente, oppure selezionare la finestra del videogame mediante ripetuta pressione della combinazione di tasti *alt-tab*.
Segue un esempio di un realistico scambio di messaggi tra un *BOT* ed il suo agente di controllo:

Agente:	telnet localhost 33333	(Connessione all'infrastruttura)
BOT:	Connected to quakeII bot	(Messaggio di benvenuto: connessione avvenuta)
A:	do walkby 100	(Comando: movimento in avanti di 100 unità)
B:	OK (do walkby 100)	(Conferma della ricezione del comando)
A:	do turnby 90	(Comando: rotazione di 90 gradi)
B:	OK (do turnby 90)	
B:	TELL STOPPED 60.0045	(Notifica del BOT: non è stato possibile completare il comando a causa di un ostacolo, il BOT si è spostato in avanti di sole 60 unità. Notare come la notifica è asincrona e può sopraggiungere in qualunque momento, in questo caso dopo l'invio di un comando successivo a quello interessato.)
B:	TELL KRYPTONITEFAR	(Notifica di un evento asincrono: il BOT ha avvistato un oggetto Kryptonite.)
A:	ask radius 100	(Richiesta: cosa si trova attorno al BOT nel raggio di 100 unità?)
B:	OK (ask radius 100) 1 player 156.5 86.9 23.9	(Risposta del BOT: conferma la ricezione del comando ed elenca gli oggetti rilevati, in questo caso solamente il personaggio controllato dall'utente.)

ESEMPIO APPLICATIVO: GUARDIE E LADRI

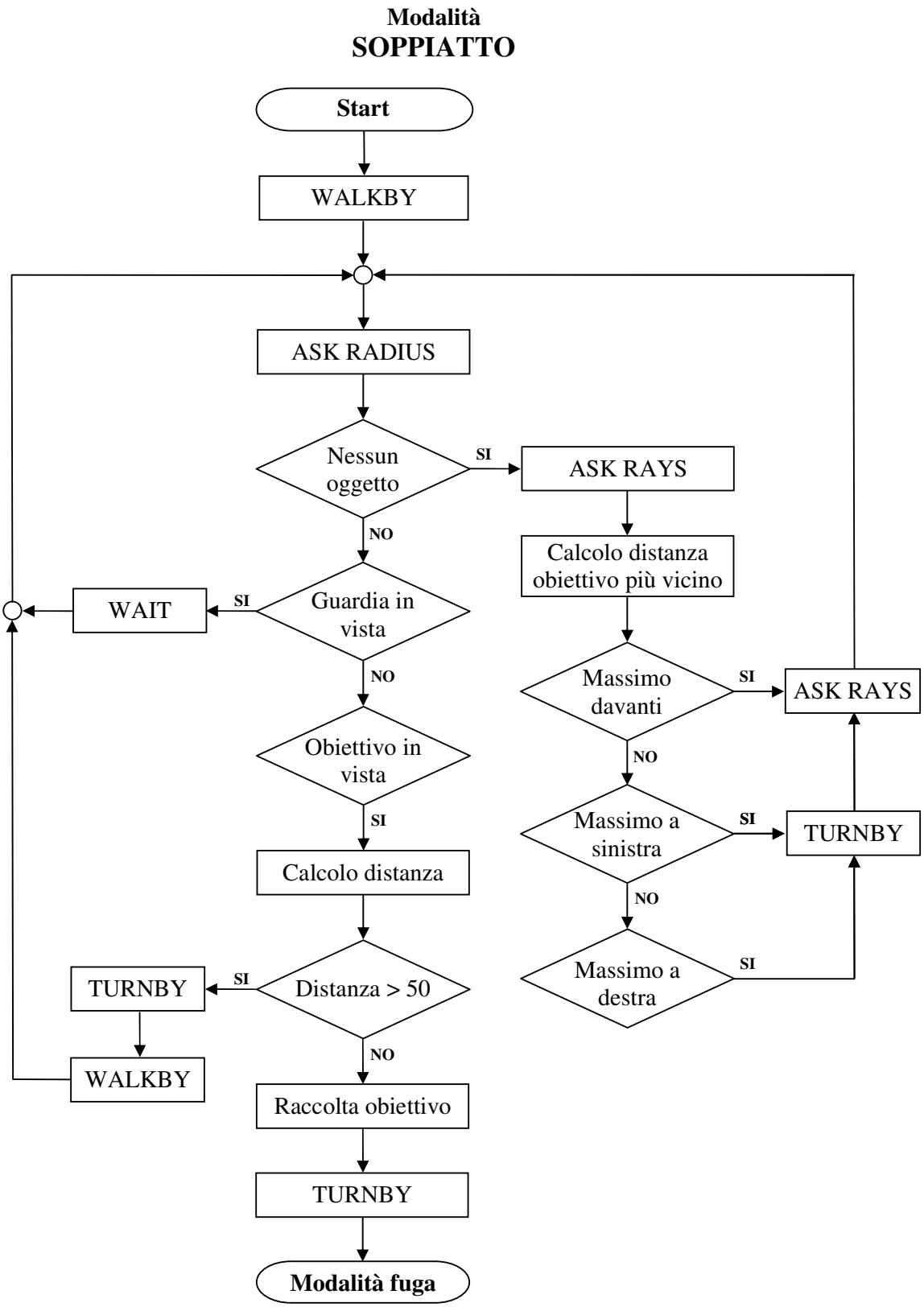
Per verificare il funzionamento dell'infrastruttura e del protocollo di comunicazione si è sviluppato un sistema multiagente di esempio ispirato al noto gioco *Guardie & Ladri*.
In particolare, nella stanza centrale della mappa di gioco *base2* viene collocato all'avvio dell'infrastruttura un oggetto di tipo *data*, rappresentante l'obiettivo che il *ladro* dovrà tentare di rubare; è stato scelto tra i numerosi disponibili proprio questo particolare oggetto in primo luogo per via del pertinente modello grafico, un cd-rom ruotante su se stesso, in secondo luogo perché è uno dei pochi oggetti la cui raccolta non influisce sulle caratteristiche fisiche del *BOT* che compie l'azione.
All'interno della mappa di gioco si trovano ad agire due *BOT*, controllati da due distinti agenti completamente indipendenti tra loro: uno di questi rappresenta il *ladro*, mentre il secondo svolge invece le funzioni della *guardia*.
Il *ladro* fa il suo ingresso nella mappa in un punto collocato a diversi corridoi di distanza dall'obiettivo e non conosce l'ambiente circostante: il suo compito è quindi quello di esplorare la mappa di gioco, cercando di raggiungere nel minor tempo possibile l'obiettivo, compatibilmente con la necessità di muoversi molto lentamente per evitare di produrre rumori che potrebbero richiamare l'attenzione della guardia; una volta raggiunto e raccolto da terra il cd-rom da trafugare, deve quindi cercare di guadagnare il più velocemente possibile l'uscita (coincidente nel nostro esempio con il punto di ingresso nella mappa) senza farsi catturare dalla guardia in allarme. Se il *ladro* riesce a trafugare il cd-rom e raggiungere l'uscita vince automaticamente la partita.
La *guardia* inizia invece la sua vita virtuale nelle vicinanze del cd-rom obiettivo e sin dal primo momento comincia ad effettuare una ronda attorno ad esso, per cercare di individuare il *ladro* che certamente tenterà di avvicinarsi. Non appena lo individua nelle sue vicinanze, gli corre incontro per cercare di catturarlo, sia che questi abbia già rubato il cd-rom sia che stia semplicemente cercando di avvicinarsi. Se la cattura riesce la *guardia* vince la partita.
A queste regole classiche del gioco ne vengono aggiunte alcune altre per tenere conto della particolari caratteristiche dell'ambiente virtuale:

- Il raggio di visione (cioè la massima distanza entro cui vengono rilevati gli oggetti collocati attorno ai *BOT*) è di 200 unità per il *ladro*, solo 100 per la *guardia*.
- Lo spazio percorso ad ogni passo di movimento è di 50 unità per la *guardia*, mentre è di sole 20 unità per il *ladro*, costretto a muoversi furtivamente.
- Perché il *ladro* possa raccogliere da terra il cd-rom obiettivo è sufficiente che si trovi in un raggio di 50 unità da esso, non necessariamente nella sua esatta posizione.
- Allo stesso modo la *guardia* è in grado di catturare il *ladro* se si trova a non più di 50 unità di distanza.
- L'infrastruttura *Quagents* non implementa alcuna forma di controllo sul numero di comandi consecutivi eseguiti dai *BOT*, come ad esempio la classica regola dell'esecuzione di un comando per *BOT* a rotazione: questo fa si che un agente più veloce degli avversari nel computare i comandi da inviare al *BOT* da lui controllato li veda poi eseguiti più velocemente di quelli inviati dagli altri.

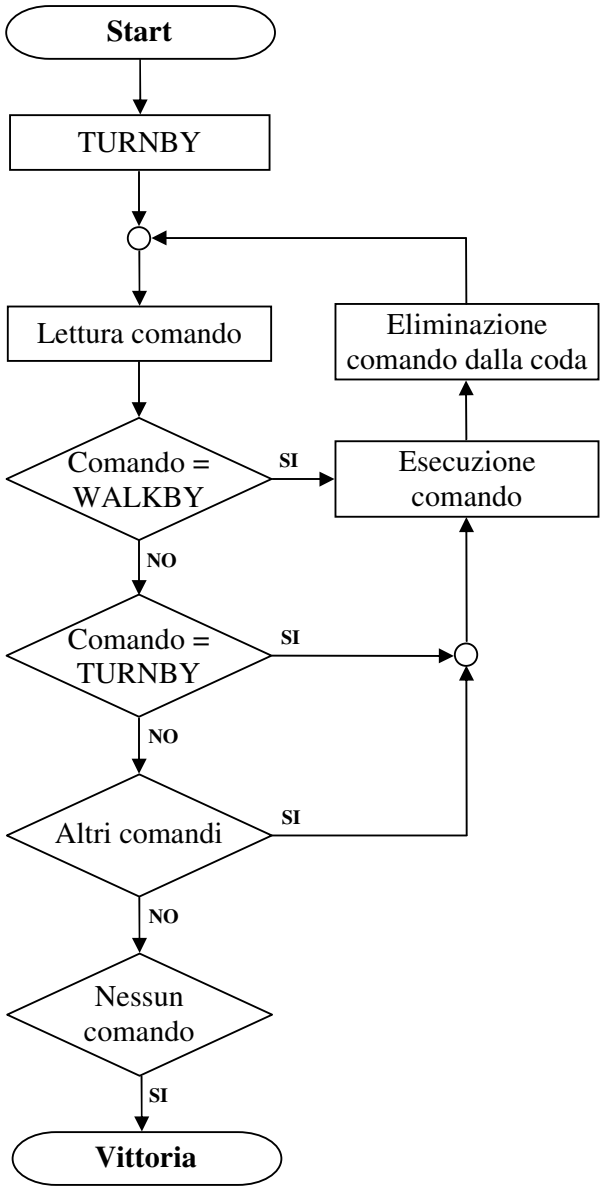
L'AGENTE LADRO

Le classi di controllo e di gestione delle comunicazioni dell'agente *ladro* sono state sviluppate in linguaggio *Java* sfruttando le potenti funzioni di gestione dei *socket* che questo linguaggio mette a disposizione, mentre la gestione del comportamento da tenere è delegata ad un programma ausiliario in linguaggio *Prolog*.
Il risultato è di fatto un agente ibrido, in cui il *cuore intelligente*, sviluppato in *Prolog*, si interfaccia attraverso grazie all'utilizzo del motore prolog *2Prolog* alle classi *Java* di gestione delle comunicazioni, attraverso di esse riceve le notifiche del *BOT*, valuta i successivi comandi da eseguire e, sempre sfruttando le classi di controllo, trasmette i nuovi comandi al *BOT*.
Una volta avviato, l'agente provvede immediatamente ad stabilire una connessione con l'infrastruttura *Quagents*, a caricare la teoria prolog utilizzata ed a mostrare a video una finestra grafica dalla duplice funzione: contiene infatti un riquadro di testo in cui vengono visualizzati all'utente tutti i comandi inviati al *BOT* e le sue risposte, e contemporaneamente mette a disposizione un pulsante per arrestare in qualunque momento l'agente.
Successivamente, il controllo passa al motore prolog, il quale valuta di volta in volta il successivo comando che il *BOT* dovrà eseguire per conseguire il suo obiettivo basandosi sulla teoria caricata e sullo storico dei precedenti comandi eseguiti.

L’algoritmo di comportamento seguito è il seguente:



Modalità
FUGA

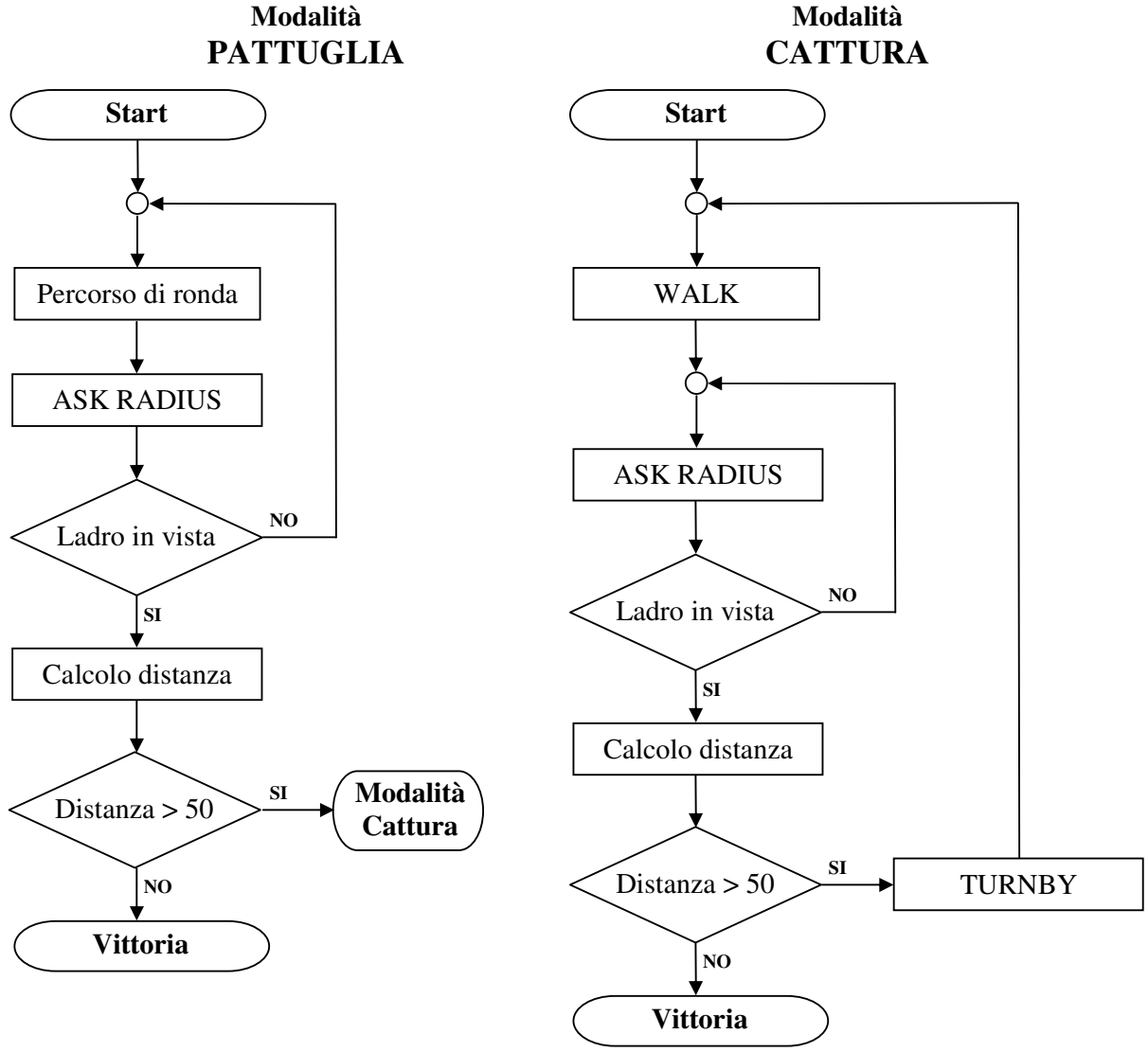


In un primo momento, il *ladro* si trova in una modalità di comportamento detta di *SOPPIATTO*, utilizzata per avvicinarsi all'obiettivo. In questa modalità il *ladro* avanza di un passo, per poi guardarsi attorno con il comando *ASK RADIUS*; se non vengono rilevati oggetti rilevanti, viene eseguito il comando *ASK RAYS* per determinare la distanza dei più vicini ostacoli lungo le tre possibili direzioni di movimento, davanti a se, a sinistra ed a destra. Sulla base delle informazioni ottenute, viene scelta come direzione in cui effettuare il prossimo passo quella che presenta la maggiore distanza dagli ostacoli, quindi, se necessario, viene effettuata una rotazione per portarsi nella direzione voluta; l'agente si muove poi di un passo nella nuova direzione e reitera il procedimento illustrato, sino a che attraverso il comando *ASK RADIUS* non vengono rilevati oggetti di interesse, in particolare l'obiettivo o la *guardia.*

In quest'ultimo caso, l'agente si arresta, interrompendo qualunque comando stesse eseguendo, ed entra in un ciclo di attesa in cui continua a sondare l'ambiente circostante mediante il comando *ASK RADIUS* sperando che la *guardia* si allontani senza scorgerlo; solamente quando il risultato del comando segnala che la *guardia* è uscita dal raggio di vista, l'agente riprende la sua avanzata.
Se invece viene rilevata la presenza dell'obiettivo, ne viene calcolata la distanza e se questa è inferiore a 50 unità, massimo raggio di raccolta del *BOT*, questo viene immediatamente rubato; in caso contrario è necessario che l'agente si avvicini di più. In base alle coordinate relative del cd-rom rispetto alla sua posizione, il *ladro* calcola quindi l'angolo di rotazione necessario per portarsi in linea con l'obiettivo, ruota dell'angolo appena determinato e riprende la sua avanzata, sino a quando non raggiunge una distanza utile per raccoglierlo.
Una volta ottenuto l'obiettivo, l'agente entra nella seconda modalità operativa, detta di *FUGA*, in cui, tralasciando ogni verifica dell'ambiente circostante per velocizzare al massimo la valutazione del comportamento da seguire, viene esaminata a partire dall'ultimo la lista dei comandi precedentemente eseguiti per portarsi sull'obiettivo: tutti i comandi di rotazione e movimento vengono eseguiti in senso inverso, gli altri vengono scartati. In questo modo l'agente ha la certezza di tornare senza errori al punto di partenza seguendo un percorso già noto tentando di raggiungere la vittoria.

L'AGENTE GUARDIA

Analogamente all'agente *ladro*, anche l'agente *guardia* si compone di due distinti componenti software, un *core interno* scritto in linguaggio *Prolog* che si occupa di valutare il comportamento che il *BOT* dovrà tenere, ed un'interfaccia *esterna*, costituita da classi *Java*, per la gestione dell'interazione tra l'agente ed il mondo esterno mediante l'infrastruttura *Quagents*.
Anche in questo caso, i comportamenti che l'agente può tenere nelle varie fasi della simulazione sono due, *PATTUGLIA* e *CATTURA*:



All'avvio, l'agente segue di *default* l'algoritmo di *PATTUGLIA*, si muove cioè nell'intorno dell'obiettivo seguendo un percorso prestabilito, costituito nel nostro caso da un quadrato di 200 unità di lato centrato sul cd-rom. Partendo da uno degli spigoli, il *BOT* impiega quattro passi di movimento alla sua velocità di 50 unità per passo per percorrere ognuno dei lati, dopodiché effettua una rotazione di 90 gradi per orientarsi nella direzione del lato successivo.

Alternandoli ai comandi di movimento, l'agente invia al *BOT* anche comandi di tipo *ASK RADIUS* per sondare la porzione di spazio visibile al *BOT* alla ricerca del *ladro*. Se non viene rilevata la sua presenza, l'agente continua la sua ronda, in caso contrario valuta la distanza dal *ladro*: se questa è inferiore a 50 unità il *ladro* si considera già catturato e la *guardia* conquista la vittoria; altrimenti l'agente entra nella seconda modalità di comportamento, detta di *CATTURA*.
In questa modalità vengono acquisite le coordinate della posizione relativa del *ladro* rispetto all'agente e, attraverso di esse, viene determinato l'angolo di rotazione richiesto per portarsi in linea con il *ladro*. L'agente provvede quindi a ruotare dell'angolo appena calcolato, avanza di 50 unità in direzione del *ladro* ed aggiorna la propria distanza dal *ladro* attraverso l'esecuzione di un comando *ASK RADIUS*. Procede quindi reiterando questa serie di azioni di avvicinamento sino a quando la distanza dal *ladro* non diviene inferiore a 50 unità, permettendone la cattura ed assicurando la vittoria.

CONCLUSIONI

Il progetto di ricerca qui presentato ha avuto come obiettivo lo studio delle funzionalità offerte dall'infrastruttura *Quagents*, culminato nello sviluppo di un sistema multiagente ispirato al gioco *Guardie & Ladri*.
Nell'esempio realizzato, due agenti indipendenti controllano due *BOT* che si muovono ed agiscono contemporaneamente all'interno di una mappa di gioco: un primo agente, il *ladro*, ha come obiettivo il rubare un prezioso oggetto rappresentato graficamente da un cd-rom, mentre il secondo agente impersona una *guardia* preposta alla difesa di questo stesso obiettivo.
Attraverso l'esecuzione di numerose simulazioni è stato possibile esaminare a fondo il comportamento degli agenti nelle più svariate situazioni che possono verificarsi all'interno della mappa di gioco ed il funzionamento del protocollo di comunicazione tra agenti e *BOT*, risultato estremamente efficace e versatile pur nella sua semplicità.
Il risultato è un giudizio fortemente positivo verso questa interessante infrastruttura, che apre la strada a profonde interazioni tra i due mondi, fino ad oggi ben distinti, dei motori grafici e dei sistemi multiagente, con interessanti possibili sviluppi sia nel mondo accademico della ricerca sull'intelligenza artificiale e dell'insegnamento, sia in quello dei videogames commerciali, dove la sostituzione di agenti intelligenti agli algoritmi di valutazione del comportamento oggi utilizzati potrebbe portare a nuove futuribili forme di intrattenimento.