POLITECNICO DI MILANO

Corso di laurea in Ingegneria Informatica

Dipartimento di Elettronica e Informazione

Sviluppo di un gioco tramite l'interazione fra un robot ed il controller Wii Remote

AIR Lab

Laboratorio di Intelligenza Artificiale
e Robotica del Politecnico di Milano

Relatore: Prof. Andrea Bonarini

Tesi di laurea di:
Antonio Bianchi
Ben Chen

Anno Accademico 2007-2008

RINGRAZIAMENTI

Ringrazio i miei genitori che, a loro modo, mi hanno sempre aiutato e supportato, i miei amici che mi hanno sopportato anche quando “mi si è visto poco in giro” e mio fratello che mi ha aiutato con i disegni.
Vorrei, inoltre, ringraziare il prof. Andrea Bonarini, che ci ha supportato in questo nostro progetto e ci ha fatto conoscere il mondo dell'AirLab.
Un ringraziamento a tutte le persone che abbiamo incontrato durante la nostra permanenza all'AirLab per il loro sostegno concreto e morale. Un “grazie” speciale va a Simone Ceriani, che ci ha fornito il codice di localizzazione, nonché dell'ottima e abbondante assistenza tecnica.
Ringrazio chi produce e diffonde tutta la buona musica che rende le mie ore davanti al pc un po' meno tristi.
Infine, grazie a tutte le persone che condividono gratuitamente il loro tempo, le loro conoscenze ed i loro file, senza di loro realizzare tutto questo, e molto altro, non sarebbe mai stato possibile.

Antonio Bianchi

感谢 / Ringraziamenti

我感谢我的父母, 因为他们为了我牺牲了很多 Ringrazio i miei genitori, che sanno amarmi più di quanto amino se stessi, 我的妹妹 mia sorella, 我的外婆 mia nonna, la mia ragazza Clo, per il suo incoraggiamento e soprattutto per il suo amore, Antonio Bianchi, “il socio” che mi ha sopportato per tutto il periodo della realizzazione della tesina, Eugenio Ferramola, un fidato amico e compagno di avventure, Alessandro Arnone, Gianni Bombelli, Siegfried Cattaneo, Elena Bartolone, la mia più cara amica, tutti coloro che lavorano in AirLab per il loro aiuto (in particolar modo professor Andrea Bonarini e Simone Ceriani), le persone che hanno contribuito allo sviluppo del free software e dell'open source, senza i quali non avremmo mai potuto realizzare ciò che abbiamo fatto, ed infine me stesso, perché mi piace davvero un sacco fare quello che faccio. :)

奔 / Ben

INDICE

1 – INTRODUZIONE

Lo scopo del nostro lavoro di tesi è quello di studiare l'utilizzo del Wii Remote, il controller principale della console Nintendo Wii, in ambito robotico. Essendo il Wii Remote (di seguito abbreviato in Wiimote) dotato di un accelerometro, di una telecamera sensibile all'infrarosso, di uno speaker, di 4 led e della possibilità di vibrare, può essere utilizzato come interfaccia uomo-computer in svariate applicazioni della robotica.
In particolare, abbiamo sviluppato un sistema di gioco (Robowii) che consiste nel cercare di colpire un bersaglio, montato su di un robot, mentre il robot stesso cerca di raggiungere una determinata posizione. Abbiamo utilizzato un robot a 2 ruote fisse parallele, controllabili separatamente.
Robowii è stato sviluppato in Linux, utilizzando il linguaggio C++. La comunicazione con il Wiimote è stata ottenuta tramite la libreria Wiiuse.
Sono stati, inoltre, utilizzati i seguenti software, precedentemente sviluppati dal Politecnico di Milano all'interno del progetto MRT (Milan Robocap Team):

- DCDT, per lo scambio di messaggi all'interno dei vari componenti di Robowii
- Mr. Brian, per la programmazione dei comportamenti del robot tramite logica fuzzy

Inoltre abbiamo utilizzato il software di localizzazione, precedentemente sviluppato per il progetto Lurch, per permettere al robot di localizzare la sua posizione all'interno del campo di gioco, utilizzando una telecamera ed appositi marker. Questo software si basa sulle librerie ArtToolkitPlus.

Di seguito viene fornita una breve descrizione delle sezioni che compongono la tesi:

"Software e Hardware utilizzato"
In questa sezione vengono descritti tutti gli strumenti hardware e software su cui il nostro progetto si basa, nonché alcuni precedenti utilizzi degli stessi e gli aspetti teorico/matematici.
"Robowii"
In questa sezione viene descritto, nel dettaglio, il nostro lavoro e la sua realizzazione. Per prima cosa viene presentata la struttura del software che gestisce le comunicazioni col Wiimote e con il robot. Successivamente sono descritti i comportamenti assegnati al robot.
"Risultati"
In questa sezione viene mostrato lo schema complessivo di gioco esemplificato con grafici che ne mostrano il funzionamento.
"Conclusioni"
In questa sezione si riassumono i risultati ottenuti, vengono descritte le problematiche riscontrate nell'attuale implementazione e i possibili sviluppi futuri al lavoro da noi svolto.

"Appendice A"
Si espone un metodo per calcolare la posizione del Wiimote in 6 gradi di libertà, utilizzando 4 LED e la libreria ArToolKit.
"Appendice B"
Si elenca il codice di configurazione da noi utilizzato per Mr. Brian.

2 – SOFTWARE E TECNOLOGIE UTILIZZATE

2.1 – Nintendo Wii Remote

Il Nintendo Wii Remote (di seguito abbreviato in Wiimote) è il principale controller per la console Nintendo Wii. Rispetto ad un tradizionale controller ha la capacità di determinare gli spostamenti a cui è soggetto tramite l'utilizzo di 2 dispositivi:

- un accelerometro ADXL330
- un sensore ottico PixArt sensibile all'infrarosso

E' alimentato con 2 batterie standard AA ed è pensato per un utilizzo wireless, comunicando tramite il protocollo Bluetooth.

Il Wiimote presenta inoltre:

- 10 tasti digitali standard, più un tasto "power" e un tasto "sync"
- uno speaker
- 4 led
- un piccolo motore elettrico interno, collegato ad una massa eccentrica, che gli consente di vibrare
- una memoria EEPROM di 16 KiB

Figura 2.1: Il controller Wii Remote

Inoltre, tramite una porta di espansione, è possibile collegare ad esso altri dispositivi, ad esempio il Nunchuk (che contiene un ulteriore accelerometro, nonché uno stick analogico e 2 tasti digitali)

Sebbene l'utilizzo principale sia quello di permettere di interagire nei videogiochi per la console Nintendo Wii, tramite l'utilizzo di varie librerie è possibile collegare il Wiimote ad un PC per ricevere i dati inviati e comunicare con esso. Questo ha reso interessante il suo utilizzo in svariate applicazioni (vedi sezione 2.3) come, ad esempio, interfacce aptiche o, in generale, modalità alternative di interazione uomo-computer. Inoltre, dato il basso costo e l'insieme di tecnologie in esso contenute, è stato utilizzato anche per lo studio di problemi di localizzazione tramite visione ed accelerometri.

Di seguito riportiamo i dettagli tecnici delle caratteristiche da noi utilizzate.

2.1.1 – Bluetooth

Il Wiimote comunica attraverso un collegamento wireless Bluetooth, tramite il chip Broadcom 2042. Le

comunicazioni seguono lo standard Bluetooth HID (Human Interface Device). Il Wiimote non richiede nessun tipo di autentificazione; per potersi connettere con esso è necessario, solamente, che sia in *"discoverable mode"*, questo avviene quando si premono contemporaneamente i tasti 1 e 2 o il pulsante sync. Una volta connesso il Wiimote invia le seguenti informazioni caratteristiche:

Name	Nintendo RVL-CNT-01
Vendor ID	0x057e
Product ID	0x0306

Il Wiimote invia "report" sul suo stato (tasti premuti, valori dell'accelerazione, ...) con una frequenza massima di 100 Hz. Di default viene inviato un "report" solo se:

- viene inviata una richiesta al Wiimote
- viene premuto un pulsante
- l'accelerometro o il sensore ad infrarossi rilevano una qualche variazione
- si connette ad esso un qualunque dispositivo esterno

E' comunque possibile impostare il Wiimote in modo da inviare, in ogni caso, un "report" ogni 1/100 di secondo. Una singola interfaccia Bluetooth può gestire più Wiimote contemporaneamente.

2.1.2 – Accelerometro

Nel Wiimote è presente un accelerometro a 3 assi ADXL330, con un range minimo di +/- 3g su ogni asse e una sensibilità del 10%. La forza rilevata su ogni asse è digitalizzata con una precisione di 8 bit. Il processo di fabbricazione dell'accelerometro rende necessario calibrare singolarmente ognuno di essi. All'interno della memoria EEPROM del Wiimote è quindi memorizzato il valore digitalizzato equivalente a 0g e a +1g su ognuno dei 3 assi.

Il funzionamento dell'accelerometro deve essere esplicitamente abilitato dopo che è stata inizializzata la connessione.

2.1.3 – Sensore ad infrarossi

Un sensore PixArt è presente nella parte anteriore del Wiimote e permette di tracciare la posizione relativa di LED ad infrarossi. In particolare il sensore funziona come una normale telecamera "pin-hole", ma rileva solamente onde elettromagnetiche nel campo dell'infrarosso (con una sensibilità massima intorno ai 940 nm). Inoltre non viene inviata l'intera immagine rilevata, ma da essa, vengono automaticamente identificati fino a 4 punti, che, corrispondono alle coordinate della proiezione sul piano immagine di particolari LED ad infrarossi. Per questo motivo solitamente si utilizza una "sensor bar", una barra che contiene 2 (gruppi di) LED infrarossi ad una distanza precisa.

Figura 2.2: Un modello di Sensor Bar, in evidenza i 2 gruppi di LED ad infrarossi

Questo metodo evita di:

- utilizzare una grande banda per trasmettere l'intera immagine rilevata dalla telecamera
- utilizzare una qualunque tecnica per l'identificazione di marker per estrapolare le coordinate sul piano immagine della barra, in quanto, in condizioni normali, i LED risultano l'unica fonte intensa di infrarossi.

Bisogna però notare che, in alcune situazioni particolari, ad esempio inquadrando lampadine ad incandescenza, i LED non risultano più l'unica sorgente di infrarossi rilevata e questo può risultare problematico.

La larghezza di campo è di circa 45 gradi in orizzontale e 35 in verticale, la risoluzione spaziale è di 1024x768, la frequenza di scansione è 100 Hz. Il sensore invia, inoltre, un'informazione (su una scala da 1 a 15) che indica, in modo approssimativo, l'intensità con cui un LED è rilevato.

Anche il sensore ad infrarossi deve essere esplicitamente abilitato dopo che è stata inizializzata la connessione con il Wiimote. Il suo eventuale non utilizzo, così come per l'accelerometro, permette di risparmiare il consumo di energia.

La “sensor bar” consente di utilizzare facilmente il Wiimote come un dispositivo di puntamento (vedi sezione 2.2.3.3.), ma non è possibile localizzare in modo esatto un Wiimote (cioè conoscere la sua posizione e il suo orientamento in 6 gradi di libertà) utilizzando semplicemente 2 LED (vedi appendice A).

2.2 – Librerie

2.2.1 – Librerie principali

Anche se le specifiche del protocollo di comunicazione del Wiimote non sono pubbliche, essendo conformi al protocollo Bluetooth HID è stato possibile, tramite reverse engineering, sviluppare facilmente svariate librerie per comunicare con esso. Di seguito riportiamo uno schema riassuntivo delle principali:

NOME	LICENZA	LINGUAGGIO	SISTEMA OPERATIVO	NOTE
Wiiuse	GPLv3 LGPLv3	C	Linux, Windows	Interfaccia a singolo thread, non bloccante.
WiiuseJ	GPLv3	Java	Linux, Windows	Bind della libreria Wiiuse in Java
DarwiinRemote	BSD	Objective C	OSX	
WiiremoteJ	Freeware (closed source)	Java	Linux, Windows, OSX	Necessita di una libreria che implementi lo standard JSR-82 (ad esempio: BlueCove o aventana)
wiigee	LGPL	Java	Linux, Windows, OSX	Necessita di una libreria che implementi lo standard JSR-82. Questa libreria permette di riconoscere il tipo di movimento eseguito impugnando il Wiimote.

2.2.2 – Wiiuse: caratteristiche principali

Abbiamo scelto di utilizzare la libreria Wiiuse, perché si è rivelata la più facile da integrare con il software di controllo del robot da noi utilizzato (vedi sezione 2.3 e 2.4.3).

Infatti la libreria Wiiuse ha le seguenti caratteristiche:

- compatibilità con Linux e con il driver Bluetooth nativo del kernel 2.6 (Bluez)
- utilizzo del linguaggio C
- utilizzo di un unico thread, non bloccante
- codice sorgente disponibile e documentato, con licenza aperta

– implementazione di tutte le funzionalità principali richieste dal nostro utilizzo

2.2.3 – Wiiuse: utilizzo

Per una guida completa delle funzioni e delle strutture dati di Wiiuse vedere la documentazione disponibile sul sito ufficiale; di seguito illustreremo solamente le principali, con particolare attenzione a quelle da noi utilizzate o che potrebbero essere utili per sviluppi futuri.

2.2.3.1 – Inizializzazione connessione e ciclo principale

Una volta compilate e installate, per utilizzare le librerie Wiiuse in ambiente Linux, usando come compilatore GCC, è sufficiente includere il file wiiuse.h e compilare il proprio eseguibile con il flag -lwiiuse.

Per inizializzare una connessione si usano le seguenti istruzioni:

```
wiimotes =  wiiuse_init(MAX_WIIMOTES);
found = wiiuse_find(wiimotes, MAX_WIIMOTES, CONNECTION_TIME);
if (!found) {
        printf ("No wiimotes found.");
                return 0;
}
connected = wiiuse_connect(wiimotes, MAX_WIIMOTES);
if (connected)
        printf("Connected to %i wiimotes (of %i found).\n", connected, found);
else {
        printf("Failed to connect to any wiimote.\n");
        return 0;
}
wiiuse_motion_sensing(wiimotes[0],1);
wiiuse_set_ir(wiimotes[0], 1);
```

La funzione wiiuse_init inizializza un array di `MAX_WIIMOTES` elementi, poi tramite wiiuse_find vengono cercati, per `CONNECTION_TIME` secondi, Wiimote in *"discoverable mode"* a cui connettersi, infine la funzione wiiuse_connect completa la connessione e restituisce il numero di Wiimote a cui ci si è connessi.

Le funzioni `wiiuse_motion_sensing` e `wiiuse_set_ir` abilitano, rispettivamente, l'accelerometro e il sensore ad infrarossi (nell'esempio solo per il primo Wiimote rilevato).

Il ciclo principale del programma va strutturato nel seguente modo:

```
while (1) {
        if (wiiuse_poll(wiimotes, MAX_WIIMOTES)) {
            /*
            *      ...
            */
        }
        wiiuse_cleanup(wiimotes, MAX_WIIMOTES);
        return 0;
}
```

E' necessario utilizzare un loop infinito in quanto le librerie non implementano nessun meccanismo ad

eventi o multithreading. La funzione `wiiuse_poll` restituisce il numero di Wiimote su cui è avvenuto un qualche evento. Dopo l'esecuzione di questa funzione la struttura dati rappresentante un Wiimote è aggiornata con il suo stato corrente.
Ad esempio:
`wiimotes[0]->event`
assumerà il valore corrispondente all'evento appena generato su di esso, oppure:

`wiimotes[0]->orient.roll`
restituisce l'angolo di roll attuale.

Di seguito supporremo di essere collegati ad un singolo Wiimote e ci riferiremo ad esso con la variabile
`struct wiimote_t* wm`

2.2.3.2 – Dati relativi all'accelerometro

Per accedere ai dati relativi alle accelerazioni sui 3 assi è sufficiente utilizzare:

`wm->accel.x` `wm->accel.y` `wm->accel.z`
Questo è il valore "raw" rilevato, il valore in termini di forza di gravità viene restituito, invece, da:

`wm->gforce.x` `wm->gforce.y` `wm->gforce.z`
Questo valore è ottenuto dalle librerie interpolando linearmente il dato "raw" con i valori di 0g e +1g memorizzati nel Wiimote.

Supponendo il Wiimote non soggetto ad accelerazioni esterne, l'unica rilevata è quella di gravità. In questo caso è possibile determinare l'orientamento del Wiimote in base a su che assi essa viene rilevata.

Riferendoci alla convenzione adottata in figura 2.3, l'angolo di roll è ottenibile con la funzione `atan2(x,z)`, quello di pitch con `atan2(y,z)` e l'angolo yaw con `atan2(x,y)` dove x,y,z sono le accelerazioni di gravità sui rispettivi assi. Queste equazioni però restituiscono valori affidabili solo sotto particolari condizioni, ad esempio, in caso di utilizzo "normale", cioè impugnando il Wiimote perpendicolarmente alla forza di gravità e con i pulsanti verso l'alto (figura 2.4 Wiimote 1), una variazione di yaw, non provoca nessun cambiamento significativo nell'accelerazione rilevata sugli assi x e y. Per questo motivo il calcolo di questo angolo risulta poco significativo e non è implementato nelle librerie. L'angolo di yaw è invece determinato con l'utilizzo del sensore ad infrarossi (vedi sezione 2.2.3.4).

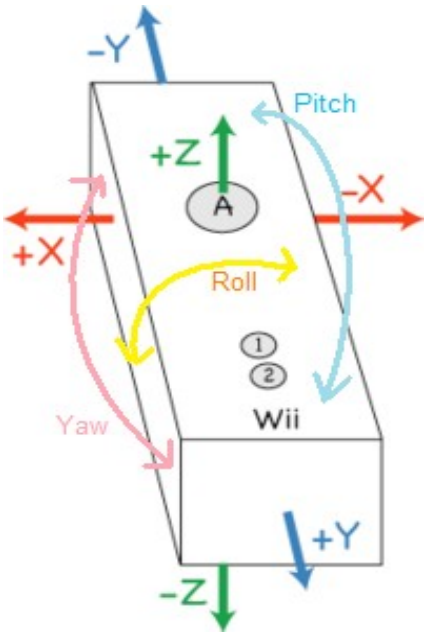


Figura 2.3: La convenzione sugli assi di traslazione e rotazione del Wiimote usata

Considerazioni analoghe valgono per l'angolo di roll e pitch nelle posizioni rappresentate rispettivamente dalle figure (figura 2.4 Wiimote 2) e (figura 2.4 Wiimote 3).

Questi valori sono accessibili dai seguenti campi della struttura dati relativa al Wiimote:

`wm->orient.roll` `wm->orient.pitch`
`wm-> orient.yaw`
E' anche possibile ottenere una versione filtrata di roll e pitch:

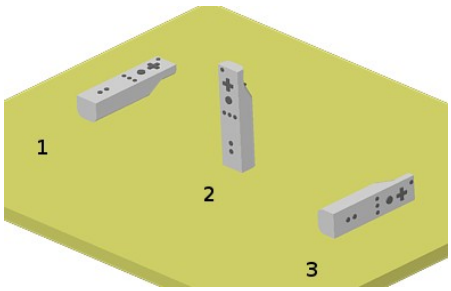


Figura 2.4: In queste posizioni un angolo di rotazione non è determinabile tramite l'accelerometro

`wm->orient.a_roll` `wm->orient.a_pitch`
E' importante sottolineare come questi dati siano significativi solo in assenza di altre accelerazioni oltre a quella di gravità, situazione determinabile verificando che la somma vettoriale delle accelerazioni rilevate sui vari assi sia, in modulo, uguale o simile ad 1.

2.2.3.3 – Dati relativi al sensore ad infrarossi

Le librerie Wiiuse restituiscono le coordinate sul piano immagine del punto in cui i LED infrarossi sono rilevati nei campi:

`wm->ir.dot[i].rx` `wm->ir.dot[i].ry`
Questi valori sono ribaltati rispetto alle coordinate effettivamente rilevati dal sensore. Utilizzando questa convenzione sono più comodi per usare il Wiimote come dispositivo di puntamento.

I campi:

`wm->ir.dot[i].x` `wm->ir.dot[i].y`
contengono i valori precedenti a cui è stato tolto l'effetto della rotazione del Wiimote, questo viene ottenuto ruotando, intorno al centro del piano immagine, i punti per un angolo opposto all'angolo di roll rilevato dall'accelerometro.

Considerando l'utilizzo normale del Wiimote, cioè puntandolo verso una barra contenente 2 gruppi di LED, i campi:

`wm->ir.ax` `wm->ir.ay`
rappresentano le coordinate del punto effettivamente puntato sulla barra dove 512 (1024/2) e 384 (768/2) sono le coordinate del centro della barra.

Questi valori sono ottenuti calcolando la media delle coordinate dei punti rilevati, dopo aver tolto ad essi l'effetto della rotazione del Wiimote (vedi figura 2.5 e 2.6).

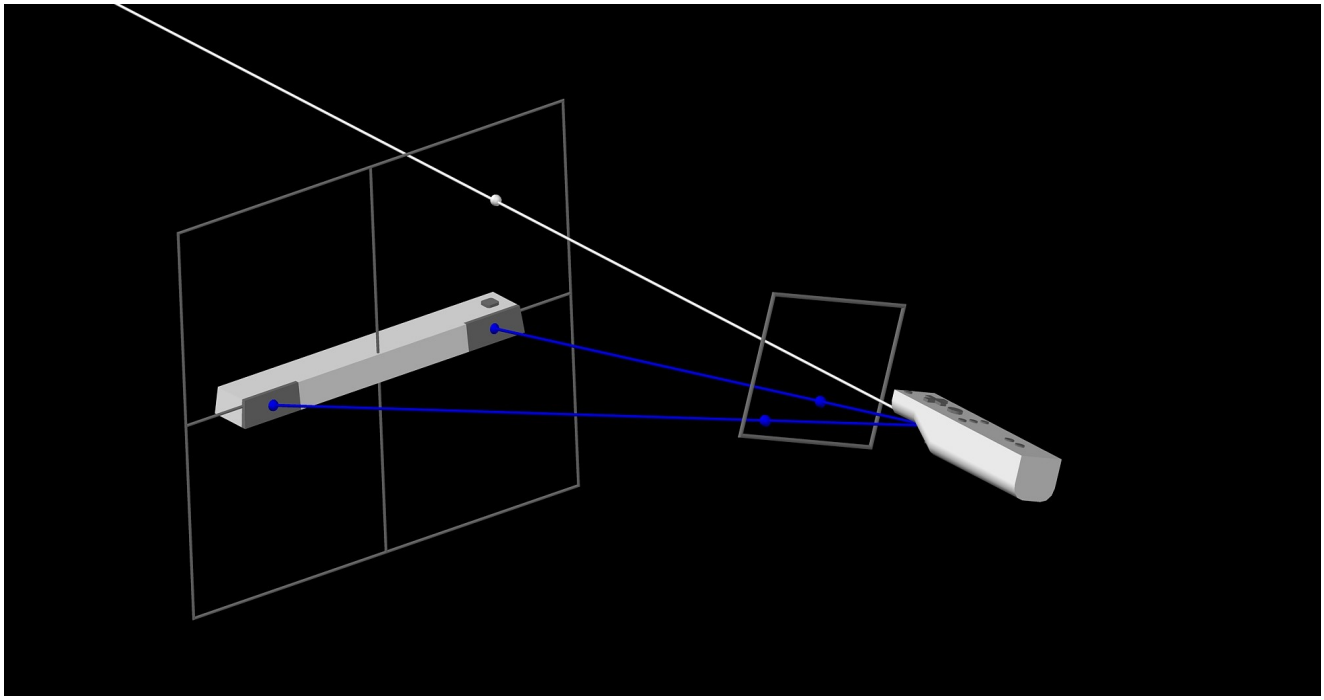


Figura 2.5: Esempio di puntamento. Il Wiimote in figura sta puntando in alto a destra rispetto ai LED, i cerchi blu rappresentano le coordinate rilevate dal sensore ad infrarossi del Wiimote

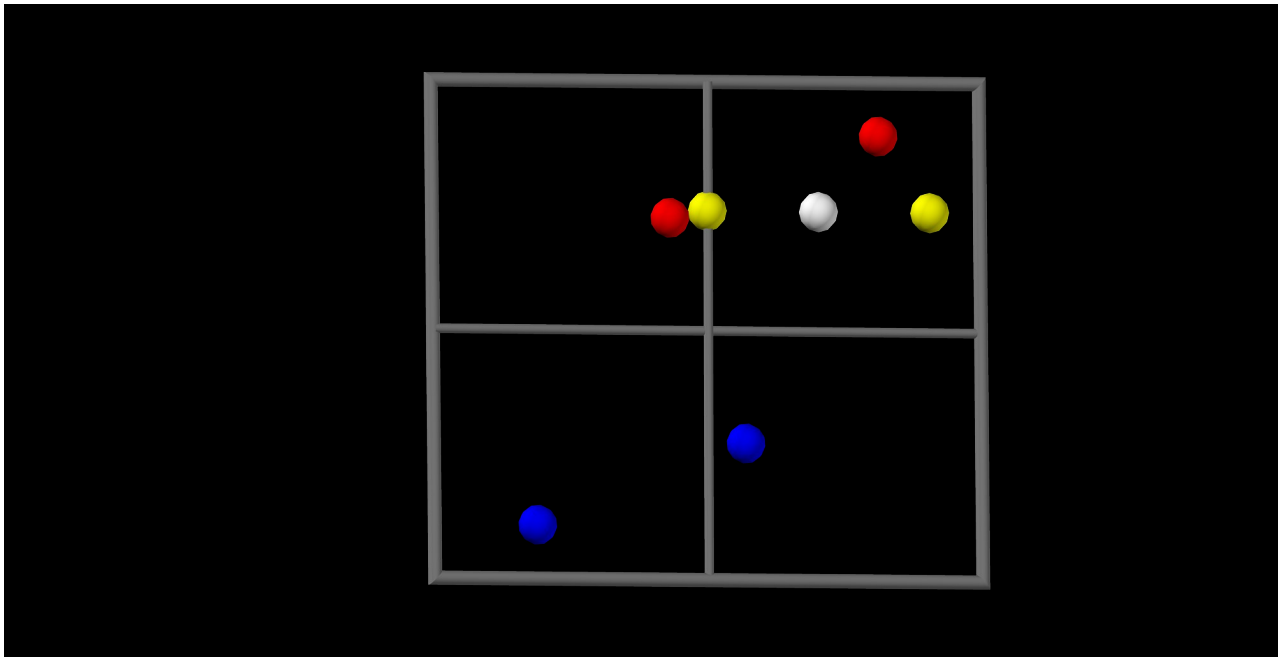


Figura 2.6: Coordinate dei LED rilevate dal sensore ad infrarossi (punti blu) e quelle restituite dalla libreria Wiiuse (punti rossi), quest'ultime risultano ribaltate in orizzontale ed in verticale. Per calcolare i punti gialli, le coordinate di quelli rossi sono ruotate intorno al centro di un angolo opposto all'angolo di roll rilevato dall'accelerometro. Per determinare il punto puntato dal Wiimote (punto bianco) è sufficiente calcolare la media delle coordinate dei punti gialli.

Nel caso che venga rilevato un solo LED, ma la libreria stima che ne esistano almeno 2 (questa condizione è verificata quando esiste un "report" precedente in cui sono stati rilevati almeno 2 LED e da esso al "report" corrente è sempre stato rilevato almeno un LED), allora il centro è calcolato come la posizione dell'unico LED rilevato a cui è sommato un valore di correzione. Altrimenti le coordinate del centro saranno semplicemente quelle dell'unico LED rilevato.

Tramite il sensore ad infrarossi è anche calcolata la distanza fra il Wiimote e la barra:

`wm->ir.z`
rappresentata con un valore indicativo su scala da 0 a 1023 ottenuto considerando che la distanza fra il Wiimote e la barra è inversamente proporzionale alla distanza sul piano immagine a cui vengono rilevati 2 LED. A partire da questo valore è calcolato, sempre in modo indicativo, l'angolo di yaw:

`wm-> orient.yaw`

2.2.3.4 Pulsanti

Le librerie Wiiuse permettono di rilevare quando un pulsante è stato premuto e rilasciato, in particolare, riferendoci al pulsante A:

`IS_JUST_PRESSED(wm, WIIMOTE_BUTTON_A)`
restituisce TRUE nel primo "report" inviato dopo la pressione del pulsante,

`IS_PRESSED(wm, WIIMOTE_BUTTON_A)`
restituisce TRUE in tutti i "report" in cui il pulsante è tenuto premuto,

`IS_HELD(wm, WIIMOTE_BUTTON_A)`

restituisce TRUE durante tutti i “report” in cui il pulsante è tenuto premuto, tranne il primo,

`IS_RELEASED(wm, WIIMOTE_BUTTON_A)`
restituisce TRUE nel primo “report” successivo al rilascio del pulsante.

2.3 – Altri utilizzi del Wiimote

Di seguito descriveremo brevemente alcuni interessanti progetti, già sviluppati, che utilizzano il Wiimote.

2.3.1 – Wiimote head tracking desktop VR display

Usando la videocamera ad infrarossi de Wiimote e due sensori LED montati sulla testa, è possibile stimare la posizione della testa. Mediante questa tecnica si può utilizzare un display per fornire immagini virtuali dipendenti dalla visione. Questo progetto è stato sviluppato da Johnny Lee, trasformando un display a due dimensioni in un portale per un ambiente virtuale ed interattivo. Il display reagisce al movimento della testa e crea sullo schermo l'illusione di una finestra reale, fornendo la sensazione di profondità e di spazio.

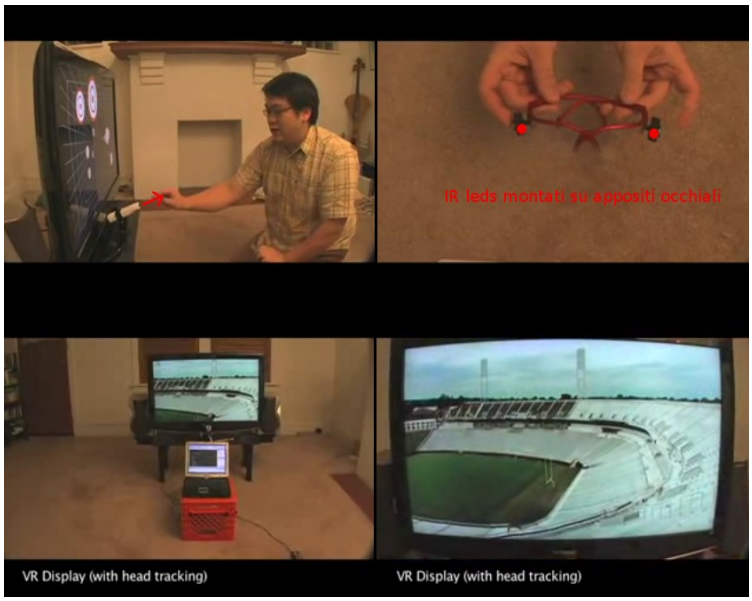


Figura 2.7

2.3.2 – Low-Cost Multi-point Interactive Whiteboards Using the Wiimote

Dal momento che il Wiimote può tracciare sorgenti di luci ad infrarossi, si può individuare la posizione di una particolare penna su cui è stata montata una sorgente IR. Puntando un Wiimote verso uno schermo di proiezione o su un display LCD, si può creare una lavagna interattiva a basso costo. Dato che un Wiimote può tracciare fino 4 punti IR, si possono utilizzare fino a 4 penne virtuali.

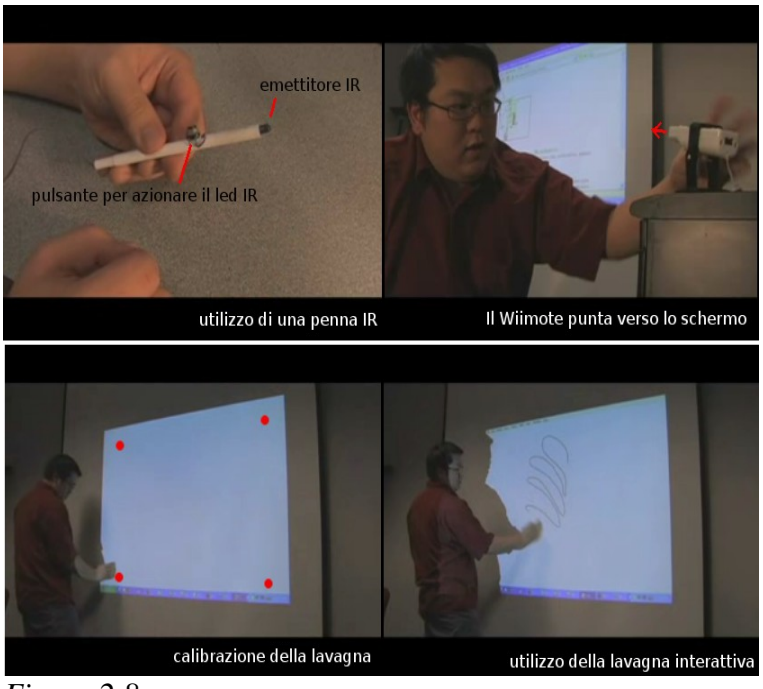


Figura 2.8

2.3.3 – Wiinstrument

Wiinstrument è una applicazione che consente di utilizzare il Wiimote con il Nunchuk per simulare una batteria elettronica. Tale applicazione sfrutta i dati rilevati dagli accelerometri per successivamente identificarne un possibile movimento di percussione e, nel caso di positività, sfrutta un driver midi per generare un suono tipico della batteria.

2.4 – Logica fuzzy e Mr. Brian

Per sviluppare i comportamenti del robot, abbiamo deciso di programmare gli stessi utilizzando regole espresse tramite logica fuzzy. In particolare ci siamo appoggiati al software Mr. Brian (Multilevel Ruling BRIAN) [1], estensione di BRIAN (BRIAN Reacts by Inferring ActioNs) [2] , sviluppato dal Politecnico di Milano all'interno del progetto MRT.

La logica fuzzy, per un'introduzione ad essa si guardi, ad esempio, [3], è una logica polivalente in cui una proposizione può avere un grado di verità compreso tra 0 e 1. La logica fuzzy trova ampio impiego in robotica, in quanto permette di programmare robot seguendo un approccio comportamentale. In questo modo, il comportamento definitivo del robot è ottenuto dalla collaborazione di vari comportamenti atomici ed indipendenti, ognuno dei quali esprime una connessione diretta fra gli input sensoriali e gli output rivolti verso gli attuatori.

Brian permette di sviluppare, rapidamente ed in modo modulare, i comportamenti che si vogliono fare assumere al robot e, nella sua estensione Mr. Brian, consente di specificare le modalità con cui essi interagiscono, essendo possibile definire anche delle gerarchie fra gli stessi. Per una descrizione completa del funzionamento di Mr. Brian si rimanda al manuale utente.

Tralasciando, inizialmente, gli aspetti legati alle gerarchie fra i vari comportamenti e riferendoci, quindi, al software Brian, il processo che, a partire dai dati di input, produce in output l'azione di controllo è il seguente:

1) Fuzzyficazione dei dati di ingresso.

2) Valutazione del valore di verità dei predicati.

3) Scelta dei comportamenti da attivare.

4) Valutazione delle regole dei singoli comportamenti.

5) Fusione dei risultati.

6) Defuzzyficazione dei risultati.

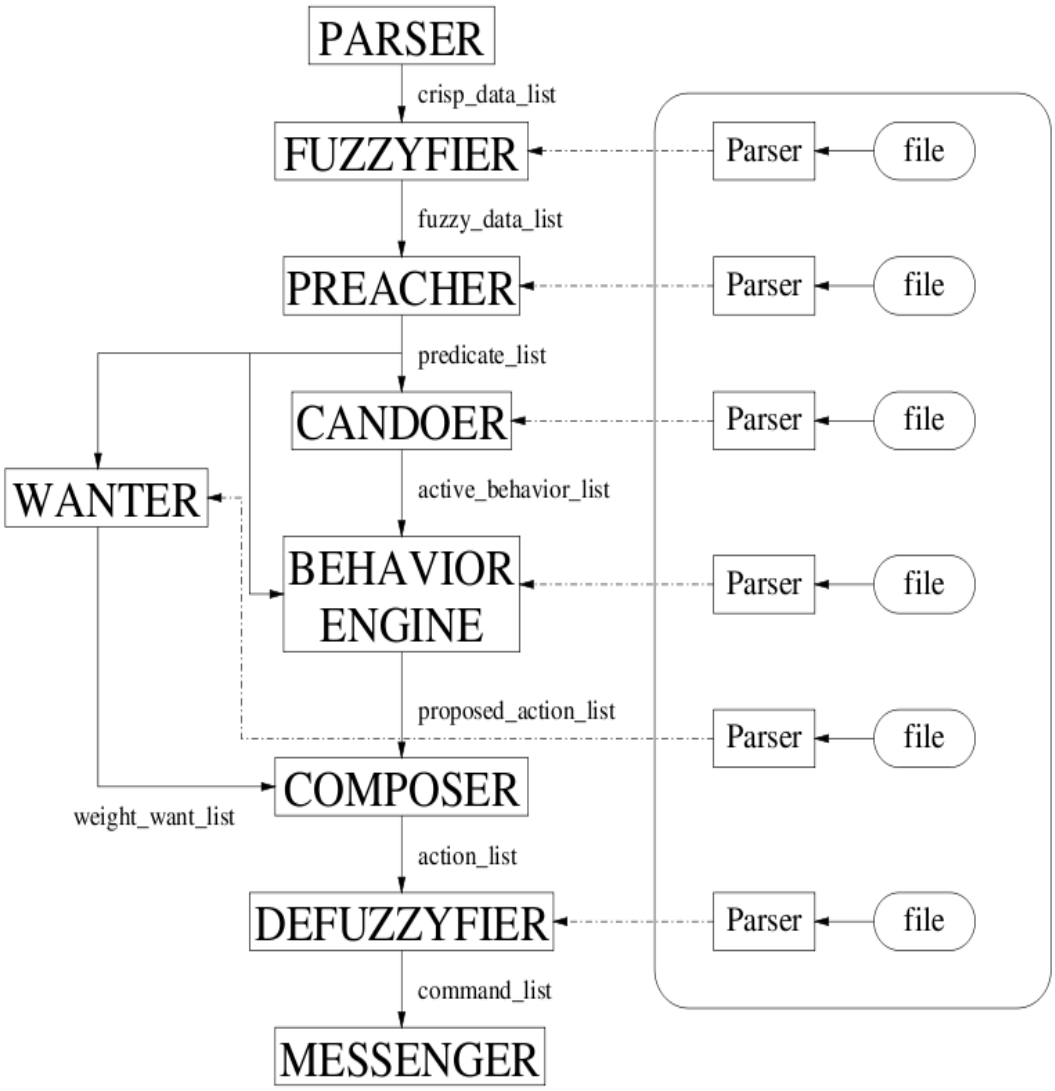


Figura 2.9: Schema di funzionamento di Brian, seguendo la terminologia utilizzata dal programma

2.4.1 – Fuzzyficazione

Per prima cosa è necessario specificare i valori di input e le tipologie di variabili fuzzy ad essi associate. Ad esempio `(DistanzaDalCentro DISTANCE)` dichiara che il dato in ingresso DistanzaDalCentro è di tipo DISTANCE.

A questo punto è necessario specificare il tipo di dato fuzzy DISTANCE. Ad esempio:

```
(DISTANCE
(TRA (CLOSE 0 0 30 50))
(TRA (NEAR 0 30 150 200))
(TOR (FAR 150 200))
)
```

definisce il tipo di variabile DISTANCE come in figura 2.10.

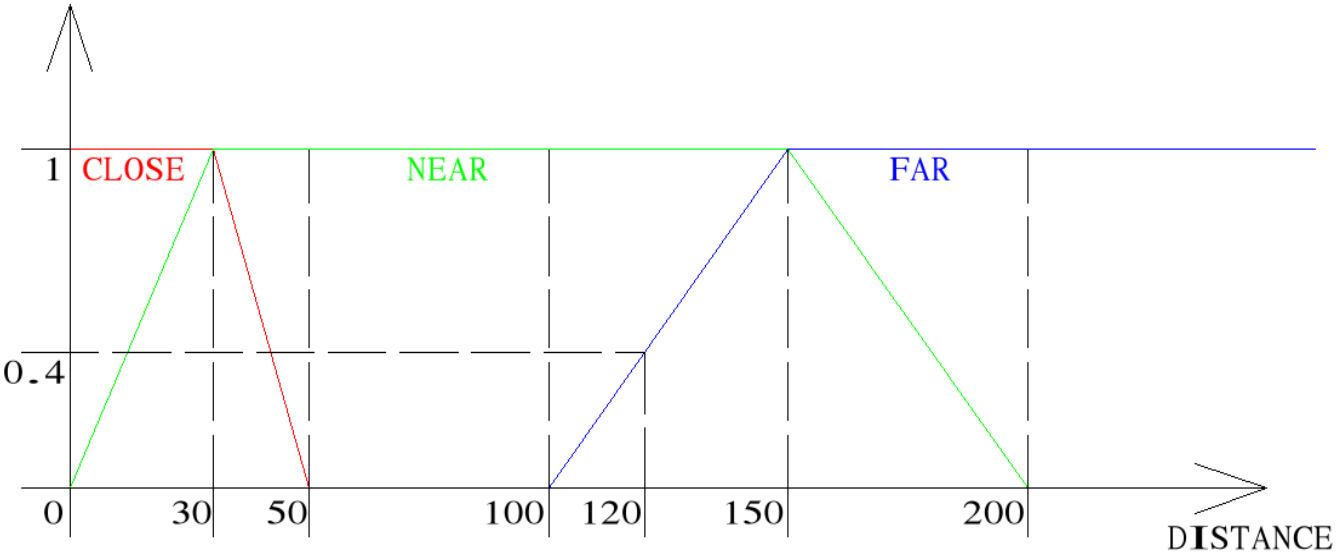


Figura 2.10: Esempio di insiemi fuzzy

In figura 2.11 sono rappresentate tutte le forme utilizzabili in Brian per definire tipi di dati fuzzy.

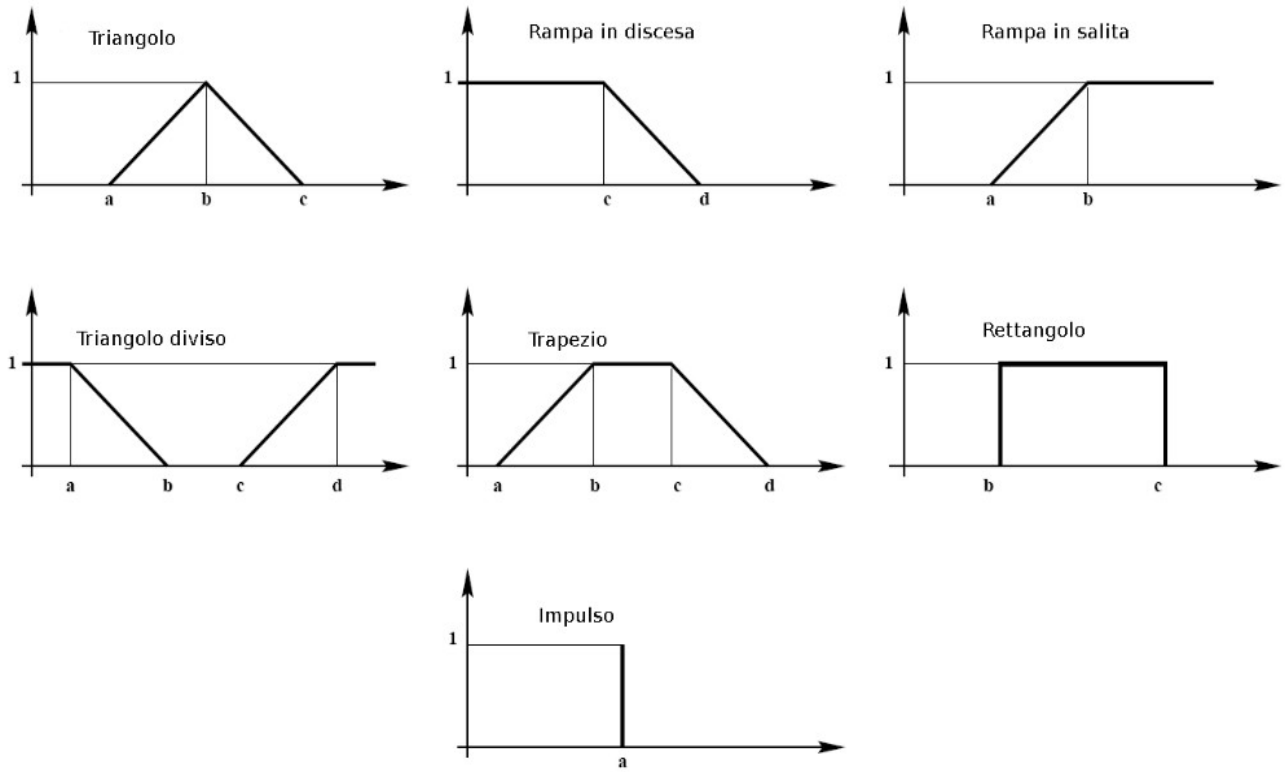


Figura 2.11: Tipi di insieme fuzzy utilizzabili in Brian

2.4.2 – Definizione dei predicati

Una volta definite le variabili fuzzy e la loro tipologia è necessario dichiarare i predicati ad esse associati. In logica fuzzy un predicato è un letterale a cui si associa un valore di verità da 0 a 1. Ad esempio:

```
DistanzaCentroFar = (D DistanzaDalCentro FAR);
```

definisce il predicato DistanzaCentroClose che avrà un valore di verità uguale al grado di appartenenza del dato DistanzaDalCentro all'interno dell'insieme CLOSE, definito precedentemente. Ad esempio, se Distanza dal centro vale 120 il suo valore di appartenenza all'insieme CLOSE è 0, all'insieme NEAR è 1, all'insieme FAR è 0.4, e quindi il predicato DistanzaDalCentro vale 0.4 (vedi figura 2.10).

E' possibile, poi, definire predicati composti a partire da altri, tramite le operazioni booleane canoniche, che in logica fuzzy sono così definite:

NOT P = 1-<P>

Q AND P = min(<Q>,<P>)

Q OR P = max(<Q>,<P>)

dove P e Q sono generici predicati, <P> e <Q> sono i loro valori.

2.4.2 – Scelta dei comportamenti da attivare (CANDO)

In Brian, tramite regole fuzzy, è possibile stabilire quali comportamenti possono essere attivati (condizioni CANDO). Ad ogni comportamento viene associato un predicato il cui valore corrisponderà al valore con cui esso dovrà essere attivato. Ad esempio:

```
VaiAlCentro = (NOT (P DistanzaCentroClose))
```

indica che il comportamento VaiAlCentro può essere attivato se la distanza dal centro non è troppo piccola.

2.4.3 – Valutazione delle regole

Dopo aver valutato i comportamenti attivi, cioè quelli il cui valore di attivazione è superiore ad una determinata soglia, Brian valuta le singole regole relative ad ognuno di essi.

Ogni regola è espressa nel seguente modo:

(*antecedente*) => (*variabile VALORE*) (*variabile VALORE*) ...

Dove l'*antecedente* è un predicato (o una composizione di essi), *variabile* è il nome di una variabile fuzzy corrispondente ad un uscita e *VALORE* è l'insieme fuzzy di valori assegnati a tale variabile.

La variabile fuzzy assume il valore indicato se il valore dell'antecedente è maggiore ad una certa soglia; alla coppia variabile-valore proposta dalla regola è associato anche il valore dell'antecedente della regola stessa che rappresenta l'importanza della regola e quindi del valore da essa generato.

Più regole possono specificare valori diversi per la stessa variabile, in questo caso, il comportamento proporrà più coppie variabile-valore per la stessa variabile e ad ognuna di esse sarà associata un'importanza diversa, in base al valore dell'antecedente della regola che l'ha generata.

2.4.4 – Fusione dei risultati

Su tutte le coppie variabile-valore proposte da tutte le regole di tutti i comportamenti attivi, viene calcolata la media dell'importanza, tale media sarà pesata sul valore di WANT associato ad ogni

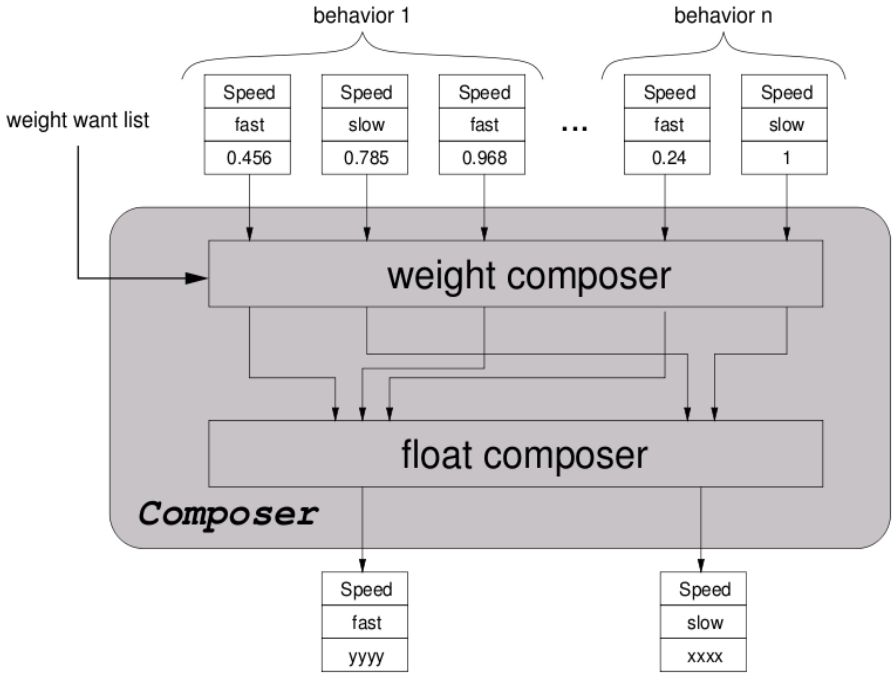


Figura 2.12: Meccanismo di fusione dei risultati

comportamento.

E' infatti possibile associare ad ogni comportamento, con la stessa sintassi delle regole CANDO, alcune regole WANT che rappresentano quando e con che intensità un certo comportamento deve essere eseguito.

2.4.5 – Defuzzyficazione

Così come i dati in ingresso (variabili reali) sono trasformati in variabili fuzzy, i valori fuzzy proposti dai vari comportamenti sono trasformati in valori reali. Brian implementa, attualmente, solo insiemi di tipo impulsivo per definire le variabili fuzzy usate come output. In questo caso il processo di defuzzyficazione consisterà semplicemente nel calcolare la media dei vari valori associati ad una variabile, pesata sull'importanza di ogni coppia variabile-valore.

2.4.6 – Mr. Brian

L'approccio di Brian permette di sviluppare i comportamenti in modo modulare e riutilizzabile. Inoltre, dividendo le condizioni CANDO da quelle WANT, permette, quando si riutilizza un comportamento in situazioni diverse, di riutilizzare le condizioni CANDO, modificando solo quelle WANT.

Questo approccio, però, non permette di gestire eventuali conflitti che possono crearsi fra comportamenti diversi, ad esempio il comportamento EvitaOstacolo potrebbe proporre di ruotare a destra, mentre, nello stesso momento, il comportamento VaiObiettivo potrebbe proporre di girare a

sinistra, compromettendo il risultato voluto dal comportamento EvitaOstacolo. Situazioni del genere potrebbero essere evitate modificando opportunamente le condizioni CANDO e WANT associate ai comportamenti, ma questo andrebbe contro il principio di sviluppo modulare degli stessi.

Questo problema viene risolto da Mr. Brian introducendo la possibilità di imporre una gerarchia ai vari comportamenti. Il ciclo di controllo di Mr. Brian risulta quindi il seguente:

1) Fuzzyficazione dei dati di ingresso.

2) Valutazione del valore di verità dei predicati.

3) Scelta dei comportamenti da attivare.

4) Valutazione delle regole dei singoli comportamenti del livello *i*.

5) Fusione dei risultati.

6) Defuzzyficazione dei risultati.

7) **se** *i* è l'ultimo livello
   allora fine
   altrimenti incrementa i, ritorna al punto 1

E' necessario specificare che i dati in ingresso al punto 1 saranno, oltre ai dati sensoriali, anche i valori in uscito al punto 6 del livello precedente e che i predicati dipendenti da essi saranno rivalutati ad ogni livello. Le regole dei vari comportamenti potranno quindi basarsi su predicati che rappresentano i valori proposti dai livelli precedenti. Inoltre possono annullare del tutto i valori proposti dai livelli precedenti.

Ad esempio il comportamento EvitaOstacolo, può annullare ogni spostamento proposto dagli altri comportamenti, se posto ad un livello superiore, con la seguente regola:

```
(OstacoloVicino) => (&DEL.VelocitàRotazione ANY)(&DEL. VelocitàTangenziale ANY);
```

2.5 – DCDT (Device Communities Developement Toolkit)

Un aspetto essenziale, sotto l'aspetto dello sviluppo e dell'implementazione di un progetto di robotica, consiste nell'integrare e far comunicare fra loro, i vari componenti che comandano il robot. Per soddisfare tale scopo è stata necessaria la creazione di un sistema base, un middleware, che riesca a incorporare i vari strumenti disponibili e che ne garantisca la loro interazione in modo semplice e dinamico.

Il software DCDT (Device Communities Developement Toolkit) [4] sviluppato all'interno dell'Airlab è un sistema orientato alla gestione degli agenti. Un agente è un'entità astratta, è rappresentata, nel codice sorgente, da classi, di tipo Expert. Nel seguito si preferisce usare la parola agente, in quanto la parola esperto descrive solamente una entità che ha determinate esperienze e capacità in certi campi applicativi, mentre un agente non solo può essere considerato un esperto, ma è anche un'entità che compie determinati servizi per conto di altri.

I vari agenti, per funzionare, hanno bisogno di eseguire i loro compiti in modo parallelo e di scambiarsi messaggi. DCDT permette di creare un sistema multiagente senza gravare l'utente degli aspetti legati alla programmazione in multithreading e della comunicazione fra thread diversi e permette di gestire con facilità la creazione e la comunicazione di più agenti, anche in esecuzione su computer diversi.

2.5.1 – Inizializzazione del Dispatcher e creazione degli agenti

La dinamica di funzionamento dello scambio dei messaggi è descrivibile con un modello di un ipotetico ufficio postale che ha il compito di ricevere, smistare e consegnare i messaggi mediante una classe chiamata Dispatcher che svolge il lavoro di postino. DCDT ha una politica pubblica/sottoscrivi che nasconde la distribuzione fisica dei messaggi. In questo modello tutti gli agenti hanno la possibilità di pubblicare messaggi inviandoli all'ufficio postale. Ad ogni messaggio è assegnato un tipo; un agente ha la possibilità di sottoscriversi ad una lista di interesse, specificando il tipo di messaggi che intende ricevere. In questo modo solo i diretti interessati possono ottenere le informazioni precedentemente richieste e non vengono, quindi, prodotte copie inutili dello stesso messaggio, aumentando l'efficienza e l'affidabilità del sistema. Nel caso in cui dei messaggi arrivano nell'ufficio postale e non esiste nessun agente che ha sottoscritto quella tipologia di messaggio, i messaggi vengono scartati.

Il file principale necessario alla creazione di un robot è kernel.cpp, del quale analizzeremo adesso i punti principali. Per una guida completa a DCDT si rimanda al suo manuale utente.

Nell'esempio sotto si illustra la possibile creazione del dcdt con inserimento di un esperto (WiimExpert).

```
// dichiarazione dell'esperto
#define WIIM
#ifdef WIIM
        #include "WiimExpert.h"
        int period=0; // in microsecondi
#endif
// definizione del file di configurazione di dcdt.
// permette di impostare se DCDT deve lavorare in locacale, su una singola macchina o su una rete, in tal caso vanno impostati i parametri TCP/IP
string dcdt_config_file("./config/dcdt.conf");

int main(int argc, char** argv)
{
```

```
        // creazione del dispatcher
        ModuleDispatch* Dispatch = new ModuleDispatch (basename(argv[0],
                                                        dcdt_config_file));
        // creazione della lista degli agenti
        std::vector < StringModuleMember * >agents;
        #ifdef WIIM
        // creazione dell'esperto WiimExpert
        WiimExpert* pWiim = new WiimExpert (Dispatch, period);
        // aggiunta dell'esperto alla lista di agenti
        agents.push_back(pWiim);
        #endif
        // assegnamento della lista degli agenti al dispatcher
        for (std::vector < StringModuleMember * >::iterator i = agents.begin ();
                                                        i != agents.end (); i++)
        Dispatch->AddMember (*i);
        // attivazione di tutti gli agenti creati
        for (std::vector < StringModuleMember * >::iterator i = agents.begin ();
                                                        i != agents.end (); i++)
                Dispatch->ActivateMember (*i);
        // attivazione del dispatcher
        Dispatch->LetsWork ();
}
```

Ogni singolo agente dovrà implementare i metodi `RunInit()` e `RunDuty()` e seguirà una struttura come quella dell'esempio seguente:

```
// costruttore
WiimExpert::WiimExpert(ModuleDispatch* kernel,int period) :
                StringModuleMember(kernel,period,"WiimExpert")
{
        cout<<"\nWiimExpert built"<<endl;
};
// distruttore
WiimExpert::~WiimExpert()
{
        cout<<"WiimExpert destroyed"<<endl;
};
// il metodo RunInit() viene eseguito al momento della creazione dell'agente
void WiimExpert::RunInit()
{
        //  nel nostro caso dichiariamo di voler ricevere
            tutti i messaggi di tipo MSG_POSITION
           AddMessageRequest( MSG_POSITION, LOCAL );

      //   ...

        cout<<WiimExpert initialized"<<endl;
};

// il metodo RunDuty() viene eseguito in modo ciclico
void WiimExpert::RunDuty()
{
      // il metodo ReceiveLastMessage riceve l'ultimo messaggio del tipo
      specificato ed elimina l'esperto this dalla lista degli esperti in attesa di
      ricezione
      if((ReceiveLastMessage(MSG_POSITION, buffer,false) > 0)){
            // ...
```

```
        }

        // ...
}
```

E' importante precisare che il metodo `RunDuty()` viene eseguito in modo ciclico, ad intervalli regolari, definiti dalla costante `period`, in fase di inizializzazione dell'agente. Se questa costante è impostata a zero l'agente sarà eseguito in modo continuo.

Il tempo di esecuzione di un agente è influenzato anche dall'istruzione di ricezione dei messaggi `ReceiveLastMessage`, infatti, se il secondo parametro è impostato `TRUE` l'esecuzione dell'agente sarà interrotta fino a quando non venga ricevuto un messaggio del tipo inviato. Se invece è impostato `FALSE` l'esecuzione andrà avanti, anche se non è disponibile nessun nuovo messaggio del tipo richiesto.

La creazione e l'invio dei messaggi saranno trattati nel prossimo paragrafo.

2.5.2 – La classe StringModuleMember

DCDT non impone nessun vincolo sul payload dei messaggi, né nessun meccanismo di creazione e parsing dello stesso. All'interno del progetto MRT è stata sviluppata un'estensione a DCDT che aggiunge alcune funzionalità per la creazione e l'invio dei messaggi. Ogni agente viene dichiarato come sottoclasse della classe `StringModuleMember`, che implementa i vari metodi per la creazione e l'invio di messaggi.

Di seguito analizzeremo i principali.

`NewMessage(TIPO_MESSAGGIO, ID_AGENTE)`

Questo metodo crea un nuovo messaggio con sintassi XML con i seguenti campi:

`<MESSAGE id: ID_AGENTE sender:INDIRIZZO_IP_AGENTE timestamp: MICROSECONDI module: NOME_AGENTE >`

Dove id, sender e module, sono riferiti all'agente che ha creato il messaggio.

In seguito è possibile aggiungere campi dati al messaggio, modificando il membro `mCurrentMessage`, ad esempio nel seguente modo:

```
mCurrentMessage<< "<C>"<<COMANDO VALORE<<"</C>\n";
```

Infine per chiudere ed inviare il messaggio si utilizzano queste istruzioni:

```
CloseMessage();
SendAllStringMessages();
```

2.6 – Localizzazione

Per localizzare la posizione del robot, abbiamo utilizzato un sistema di fiducial marker, precedentemente sviluppato dal Politecnico di Milano per il progetto [5], che, a sua volta, fa uso delle librerie ArToolKitPlus per identificare e localizzare la posizione dei marker nell'immagine fornita dalla telecamera.

Questo sistema ricava la posizione e l'orientamento del robot, rilevando la posizione relativa del robot rispetto ad una serie di marker che nel nostro caso sono stati posizionati sul soffitto.

La procedura di localizzazione è composta principalmente dai seguenti passi:

- acquisizione dell'immagine dalla telecamera (libreria libraw1394)
- identificazione e riconoscimento dei marker nell'immagine acquisita (ArToolKitPlus)
- calcolo della posizione relativa fra 1 marker visualizzato ed il robot in 6 gradi di libertà (ArToolKitPlus)
- calcolo della posizione relativa fra 1 marker visualizzato ed il robot in 3 gradi di libertà (modulo VisionExpert)
- calcolo della posizione del robot in coordinate assolute e filtraggio (modulo VisionExpert)

2.6.1 – ArToolKit ed ArToolKitPlus

ArToolKit è una libreria sviluppata per applicazione di realtà aumentata (argumented reality), dove sull'immagine proveniente dalla telecamera, viene disegnato un oggetto tridimensionale, in corrispondenza di appositi marker. L'oggetto sarà disegnato sull'immagine del mondo reale, proprio come se si trovasse nella posizione in cui viene rilevato il marker, cioè con la stessa posizione ed orientamento del marker.

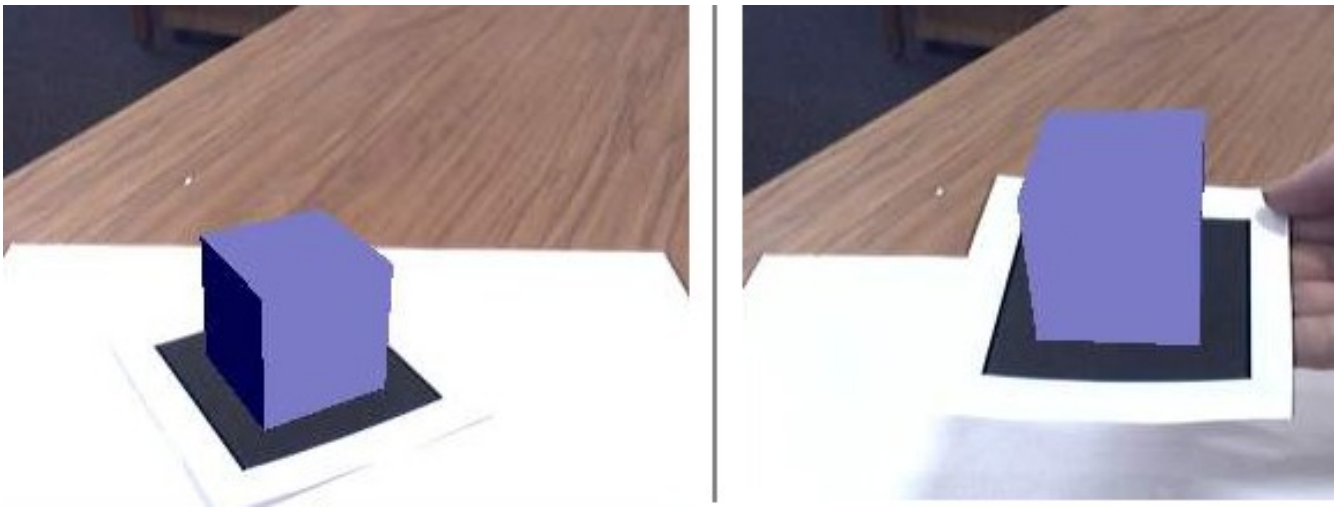


Figura 2.12: Nell'esempio ArToolKit disegna l'oggetto tridimensionale (un cubo) nella stessa posizione ed orientamento del marker.

Per fare questo ArToolKit:

- individua e riconosce (nel caso ce ne fossero più di uno) i marker nell'immagine inviata dalla

telecamera

- calcola la posizione e l'orientamento relativo fra la telecamera ed i marker
- disegna sopra i marker, con posizione e orientamento corretti un oggetto tridimensionale (vedi figura 2.12).

Di queste funzionalità solo le prime 2 sono state utilizzate dalla nostra applicazione.

I marker utilizzati da ArToolKit sono formati da un quadrato con un bordo nero, contenente un'immagine ben definita. Oltre ad alcuni marker standard è possibile registrare, tramite un'apposita procedura, nuovi marker, che verranno poi riconosciuti dalla libreria.

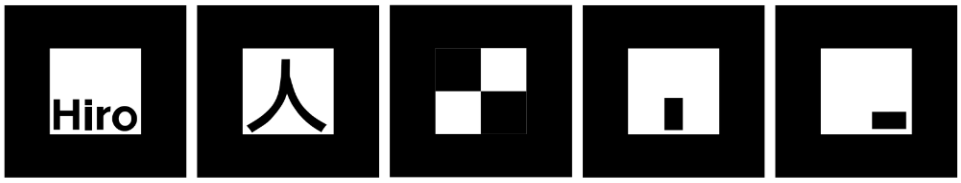


Figura 2.13: Marker di base utilizzati da ArToolKit

Nella nostra applicazione, però, abbiamo utilizzato un'evoluzione della libreria ArToolKitPlus che introduce alcune migliorie ed utilizza marker di tipo diversi. I marker utilizzati in questo caso hanno un contenuto predefinito che li identifica e non è quindi possibile memorizzarne di personalizzati.

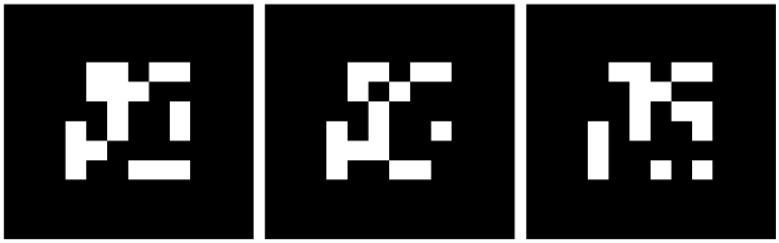


Figura 2.14: Tipologia di marker utilizzata da ArToolKitPlus

Questa libreria offre alcuni vantaggi:

- l'utilizzo di marker predefiniti garantiscono un meccanismo più rapido e preciso, rispetto ai metodi di correlazione usati da ArToolKit, per riconoscere e distinguere i marker
- migliore meccanismo di soglia in grado di riconoscere i marker anche in condizione di illuminazione fortemente disomogenea
- compensazione del fenomeno di vignettatura (zone scure sul bordo dell'immagine) che può presentarsi con alcune telecamere
- semplificazione del processo di calibrazione della telecamera, compatibile con vari tool esterni (nel nostro caso è stato utilizzato lo strumento predefinito delle librerie OpenCV)
- utilizzo di un algoritmo di stabilizzazione della stima della matrice di rototraslazione (Robust Planar Pose Tracking [6]) fra la telecamera ed il marker

2.6.2 – Calcolo della matrice di rototraslazione

Le librerie ArToolKitPlus, tramite la funzione `ArGetTransMat` restituiscono la matrice di

rototraslazione del marker rispetto alla telecamera. Il procedimento utilizzato è descritto in [7]. I parametri principali in ingresso a questa funzione sono:

- una struttura dati rappresentante il marker individuato, con i seguenti campi principali:
  - un numero identificativo del marker
  - l'intervallo di "confidenza" con cui il marker è stato rilevato
  - le coordinate sul piano immagine degli spigoli del marker
  - le equazioni delle rette passanti per gli spigoli del marker
  - l'orientamento del marker (più precisamente, quale spigolo è attualmente quello con coordinata x minore nell'immagine)
- la matrice di calibrazione della telecamera

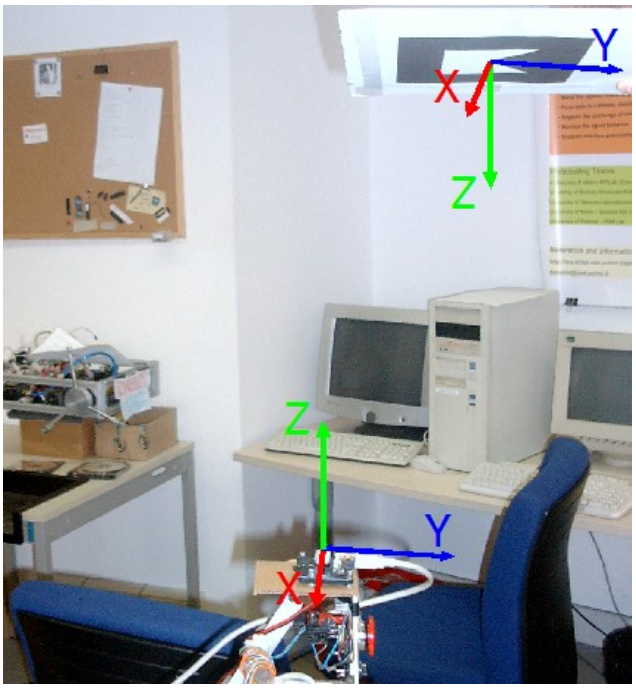


Figura 2.15: Le convenzioni usate per il sistema di riferimento del marker e della telecamera. Per quanto riguarda le rotazioni con α si intende la rotazione intorno all'asse z, con β quella intorno a y e con γ quella intorno a x

Dalla matrice di rototraslazione MT restituita è possibile, calcolandone l'inversa $TM=MT^{-1}$, ricavare la matrice di rototraslazione della telecamera rispetto al marker e da essa estrarre le coordinate e l'orientamento (in termini, ad esempio di rool, pitch, yaw) del robot rispetto al marker. Mentre la matrice MT restituisce valori stabili, la sua inversa TM risulta essere molto instabile. Il motivo geometrico alla base di questo problema è che un piccolo errore nel calcolo della rotazione intorno agli assi x e y (fare riferimento alle convenzioni in figura 2.15) nella matrice MT porta ad un grande errore nel calcolo della posizione del robot, soprattutto se esso è posto, come nel nostro caso, ad una distanza elevata dal marker. Inoltre, proprio questi valori, sono i più difficili da stimare dall'immagine rilevata del marker, in quanto una differenza, anche di pochi pixel, porta a determinare orientamenti del tutto diversi del marker intorno all'asse x e y (angoli β e γ) (vedi immagine 2.16).

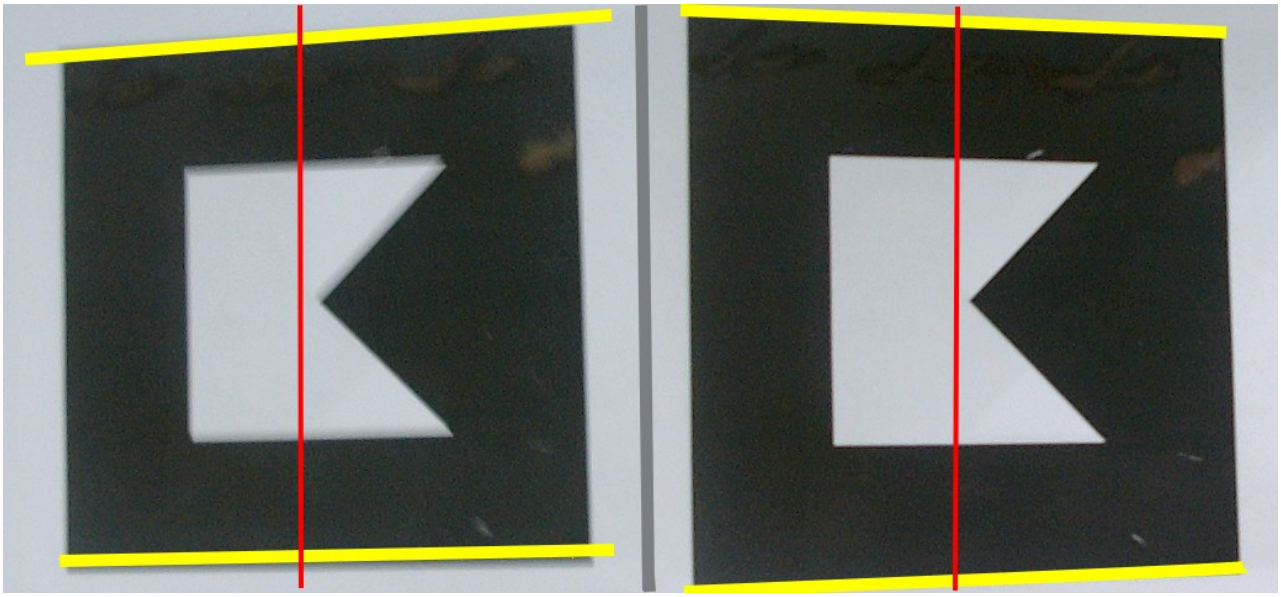


Figura 2.16 : I 2 marker sono ruotati intorno all'asse rosso di circa +30 gradi il primo e -30 gradi il secondo. Nonostante questo, le differenze fra le immagini non sono molto consistenti.

Bisogna però considerare che, nel nostro contesto applicativo, il robot si muoverà unicamente lungo gli assi x ed y e subirà rotazioni solamente attorno all'asse z. Il valore della traslazione lungo z, così come quello delle rotazioni intorno β e γ può essere considerato costante per ogni marker.

Per questo motivo durante la fase di calibrazione del sistema (vedi sezione 2.6.3) questi parametri saranno stimati, per ogni singolo marker, e ritenuti fissi durante i movimenti del robot. Il calcolo delle coordinate del robot rispetto al marker non sarà più effettuato calcolando l'inversa della matrice MT, ma della matrice MT*, ottenuta sostituendo i valori β e γ e quello della traslazione lungo z predeterminati durante la procedura di calibrazione.

2.6.3 – Calibrazione dei marker

Prima di poter utilizzare il sistema di localizzazione è necessario effettuare la sua calibrazione. Questa procedura ha 2 scopi principali:

- stimare le posizioni relative (rappresentate come matrici di rototraslazioni) fra ogni marker
- stimare le posizioni relative fra ogni marker ed un determinato sistema di riferimento assoluto
- stimare i parametri di rotazione intorno all'asse x ed y e delle traslazione z fra ogni marker ed il robot

La procedura è composta dalle seguenti fasi:

1) Per prima cosa si imposta il sistema di riferimento assoluto. Per fare questo si sceglie un marker come principale e si imposta la matrice di rototraslazione che porta da esso al sistema di riferimento assoluto.
2) Si registra un filmato ottenuto spostando il robot in modo da inquadrare, durante il suo percorso, tutti i marker, cercando di privilegiare quelle posizioni dalle quali è visibile più di un marker contemporaneamente.

3) A partire da questo filmato il sistema di calibrazione calcola i parametri β, γ e z, per ogni marker, come media dei valori determinati in tutti i frame in cui, lo specifico marker, è stato inquadrato.
4) Per ogni coppia di marker m1-m2 è calcolata la matrice di rototraslazione MM_{12} che porta dal marker m1 a quello m2. Anch'essa viene determinata come media delle matrici di rototraslazione fra la specifica coppia di marker, calcolate in ogni frame.
5) Viene creato un grafo i cui nodi sono i marker rilevati e gli archi il numero di frame in cui ogni coppia è stata rilevata contemporaneamente. Utilizzando questo grafo è possibile ottenere la matrice di rototraslazione MA_i che porta da ogni marker al sistema di riferimento assoluto. Per ogni marker si determina il percorso ottimale che porta da esso al sistema di riferimento assoluto: la matrice di rototraslazione sarà quindi ottenuta dal prodotto di tutte le matrici MM_{ij} lungo il percorso. Per determinare il percorso ottimale vanno privilegiati i percorsi con numero minore di archi e fra di essi quelli con maggior peso sugli archi. In questo modo si sommano il meno possibile gli errori di stima delle matrici MM_{ij} e si scelgono quelle matrici MM_{ij} che sono, statisticamente, più precise, in quanto calcolate su un numero maggiore di frame.

2.7 – Robot utilizzato

Il Robot da noi utilizzato è un robot a 2 ruote parallele fisse (figura 2.17). Le 2 ruote sono mosse da 2 motori DC Maxon alimentati con 4 batterie al piombo da 6 Volt e 4.5 Ah l'una, collegate in serie. Ai motori sono collegati 2 encoder HEDS 5640. Il robot può raggiungere una velocità massima di circa 1 m/s.

Figura 2.17: Modello di telecamera da noi utilizzato

Per la localizzazione è stata montata, sull'asse di rotazione del robot, una telecamera Unibrain Fire-I 1.2, con interfaccia Firewire 400 ed una risoluzione di 640x480 a 30 fps.

La scheda di controllo da noi utilizzata, chiamata AirBoard, è stata sviluppata dal Politecnico di Milano. La scheda è alimentata dalle batterie e a sua volta alimenta i motori, regolando la velocità di rotazione delle ruote tramite segnali di potenza in PWM. La regolazione della velocità si avvale di un controllore PID che ha in ingresso la velocità di rotazione delle ruote desiderata e quella rilevata dagli encoder. Sull'Airboard è presente anche un controllore fuzzy, da noi non utilizzato.

Le comunicazioni verso l'AirBoard avvengono attraverso una porta seriale RS-232 DB9, tramite la quale è possibile inviare la velocità desiderata per ognuna delle 2 ruote, inoltre, tramite un apposito comando, l'AirBoard restituisce il numero di Tick misurati dagli encoder (dall'ultima richiesta), nonché la velocità (espressa in termini di Tick al secondo).

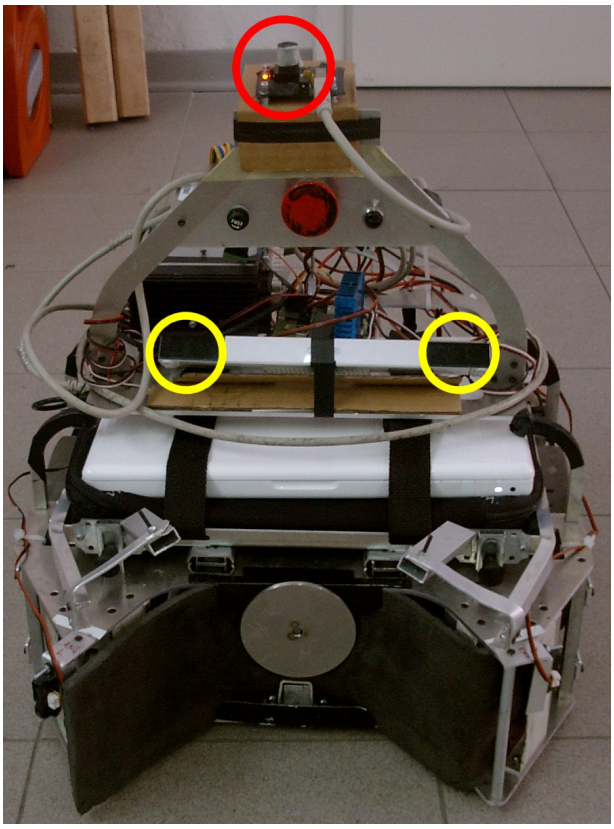


Figura 2.18: Il robot da noi utilizzato, in evidenza la posizione dei 2 gruppi di LED (cerchi gialli) e quella della telecamera (cerchio rosso)

3 ROBOWII

3.1 – Struttura di gioco

3.1.1 – Descrizione

Il gioco da noi implementato consiste nel tentare di colpire un obiettivo posto sul robot puntandolo con il Wiimote e premendo il tasto B. Per ottenere il punto del robot puntato dal Wiimote, il software si basa sui dati provenienti dal sensore ad infrarossi. Infatti l'obiettivo da colpire è posizionato al centro di una barra contenente 2 (gruppi di) LED. E' quindi possibile capire se il Wiimote è puntato verso l'obiettivo in base alla posizione sul piano immagine in cui i LED sono rilevati dal Wiimote stesso come descritto nella sezione 2.2.3.3. Nel caso in cui l'obiettivo venga colpito si totalizza un punto.

Il robot, invece, per totalizzare un punto, deve, partendo da una posizione iniziale fissata, raggiungere e mantenere per 3 secondi una posizione determinata sul terreno di gioco. Contemporaneamente, però, cercherà di evitare di essere puntato dal Wiimote.

Quando l'umano o il robot totalizzano un punto il gioco si ferma e il robot ritorna nella posizione iniziale, a quel punto, dopo 3 secondi, il gioco riprende.

Per rendere più interessante il gioco è stato implementato un meccanismo di “carica”. Non è infatti possibile colpire il bersaglio se prima non lo si è puntato, anche in modo approssimativo, per un tempo sufficiente. Questo è realizzato incrementando una variabile quando il Wiimote percepisce (in una posizione qualunque) i 2 LED della barra, e decrementandola quando non li rileva. Il robot verrà considerato colpito solo se, quando viene premuto il tasto B, il livello di “carica” sarà sufficientemente alto, altrimenti il livello di carica andrà a 0.

3.1.2 – Pseudocodice

Lo schema complessivo di gioco risulta il seguente:

```
se (TornaInPosizioneIniziale=TRUE){
      se (Posizione=POSIZIONE_INIZIALE e TempoTrascorso > 3){
            TornaInPosizioneIniziale=FALSE;
      }
}altrimenti{
      se (i LED sono inquadrati){
            aumenta carica
      }altrimenti{
            diminuisci carica
      }
      se (è stato premuto il tasto B){
            se ((il bersaglio è puntato correttamente) e (carica > SOGLIA)){
                  PunteggioGiocatore=PunteggioGiocatore+1
                  TornaInPosizioneIniziale=TRUE
```

```
            }
            carica=0
      }
}
se (Posizione=OBIETTIVO_ROBOT e TempoTrascorso>3){
      PunteggioRobot=PunteggioRobot+1
}
```

3.1.3 – Altre caratteristiche

Per aumentare l'interattività del gioco abbiamo aggiunto anche le seguenti funzionalità:

- Il livello di “carica” viene visualizzato sul Wiimote accendendo in sequenza i LED
- Quando viene totalizzato un punto il Wiimote vibra per 2 secondi e i LED mostrano la differenza di punteggio fra il giocatore ed il robot
- Quando il gioco riprende una leggera vibrazione avverte il giocatore
- Quando il giocatore preme il tasto B, senza però colpire il robot, avverte una leggera vibrazione

3.2 – Moduli software sviluppati

Lo sviluppo del software di controllo del robot, ci ha portato allo sviluppo di vari moduli, che abbiamo poi integrato e fatto comunicare fra loro tramite il software DCDT. Di seguito, con il termine modulo, si intenderà un agente, come quelli descritti nella sezione 2.5, e tutti i file contenenti funzioni ed oggetti utilizzati da esso.

I moduli da noi sviluppati sono i seguenti:

- **WiimExpert**: gestisce le comunicazioni con il Wiimote e il ciclo principale di gioco
- **BrianExpert**: in base ai dati ricevuti da WiimExpert stabilisce il comportamento che il robot deve avere, tramite la valutazione di regole in logica fuzzy (utilizzando Mr. Brian).
- **MotorExpert**: invia i comandi, ricevuti in termini di velocità angolare e tangenziale, alla scheda di controllo, invia inoltre i dati relativi all'odometria
- **PositionDummyExpert**: calcola la posizione del robot, basandosi sui comandi in arrivo BrianExpert. E' usato per verificare l'efficacia dei comportamenti in assenza del robot.
- **VisionExpert**: calcola la posizione del robot analizzando le immagini provenienti dalla telecamera, identificando marker in posizioni note
- **PositionExpert**: calcola la posizione del robot utilizzando i valori provenienti da VisionExpert e, nel caso che questi non siano disponibili, utilizza i dati di odometria provenienti da MotorExpert
- **LogExpert**: visualizza la posizione del robot, e la posizione dei LED ad infrarossi rilevata dal Wiimote

In figura 3.1 è rappresentato uno schema riassuntivo di come essi interagiscono fra di loro.

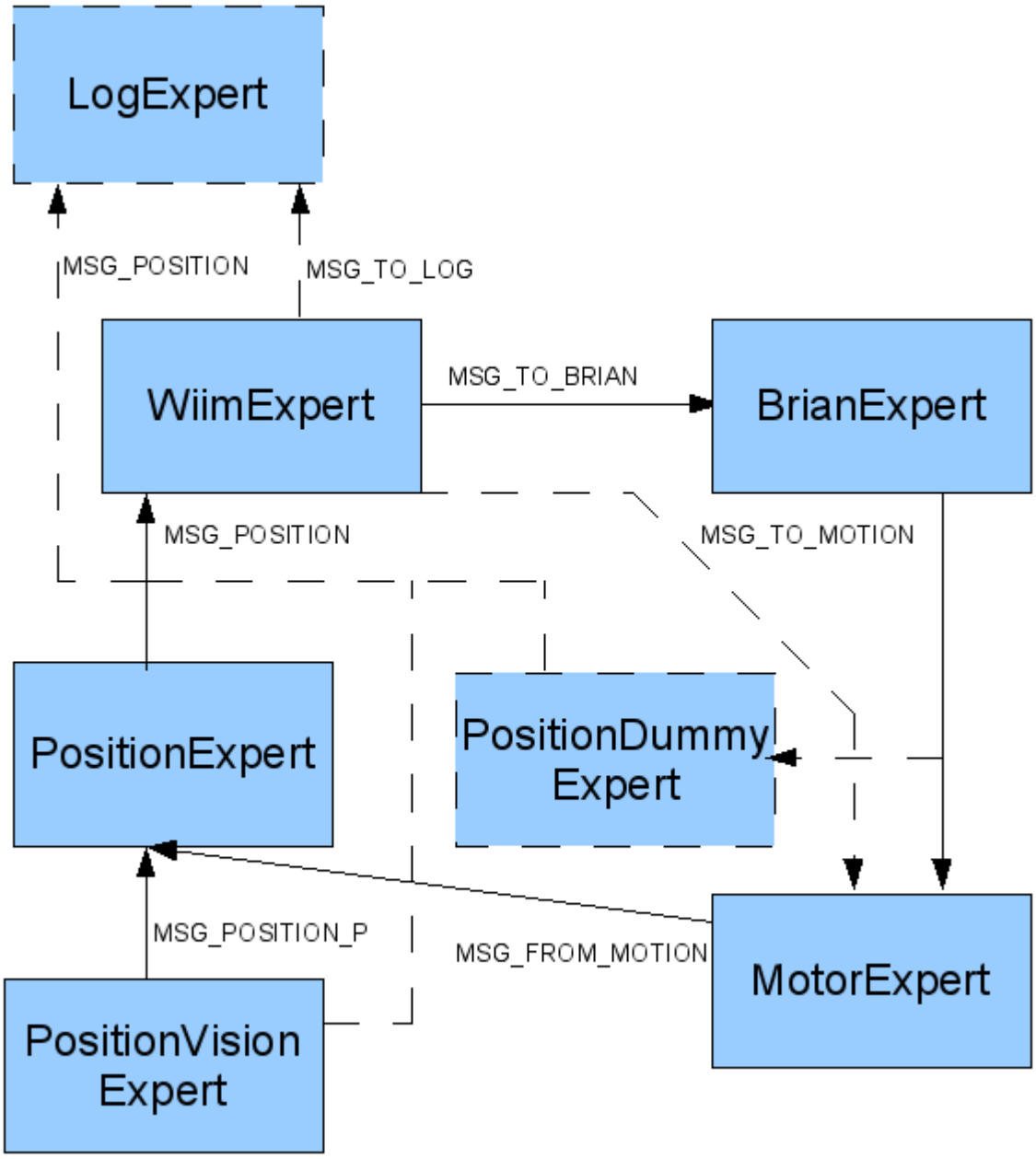


Figura 3.1: Schema riassuntivo di tutti i moduli da noi utilizzati, con i relativi messaggi. I moduli ed i messaggi tratteggiati sono quelli sviluppati solo per necessità di sviluppo e messa a punto

Specifichiamo subito che l'esistenza di 3 moduli diversi per la gestione della posizione è dovuta principalmente ad esigenze di sviluppo e messa a punto del sistema il quale, nella sua versione finale, utilizzerà solo il modulo PositionExpert per determinare la posizione del robot.

Di seguito verranno descritti nel dettaglio.

3.2.1 – WiimExpert

Il modulo WiimExpert gestisce le comunicazioni con il Wiimote, nonché la sessione di gioco, rappresenta quindi il cuore di tutta l'applicazione. Il suo periodo è stato impostato a 1000 microsecondi.

3.2.1.1 Comunicazione con il Wiimote

Una volta creato, il modulo prova a connettersi con un Wiimote ed entra in ciclo fino a che non ne trova uno disponibile. Una volta trovato un Wiimote la connessione ad esso viene segnalata facendo lampeggiare i LED dello stesso. Il modulo, a questo punto, si occupa, anche, di leggere i valori a noi utili relativi al Wiimote (come, ad esempio: i tasti premuti, i dati del sensore ad infrarossi, ...). Per calcolare le coordinate del punto puntato dal Wiimote, si fa riferimento ai campi:

```
wm->ir.ax        wm->ir.ay
```

Questi valori sono filtrati applicando ad essi una media mobile sugli ultimi 15 valori e considerando il punto non più valido, solo quando non viene rilevato nessun LED per 15 valori consecutivi. La media è pesata in modo tale da dare maggior importanza agli ultimi valori rilevati.

L'accelerometro ed il sensore ad infrarossi sono attivati subito dopo la connessione al Wiimote, inoltre viene settato un apposito flag nella configurazione del Wiimote:

```
wiiuse_set_flags(wm, WIIUSE_CONTINUOUS)
```

Impostando questo il flag, il Wiimote invierà “report” ad una frequenza di 100 Hz, sia se su di esso è rilevato un cambiamento di stato (ad esempio viene mosso), ma anche se non viene rilevato nulla. Questa scelta si è resa molto utile: in primo luogo fa si che tutti i segnali in arrivo dal Wiimote vengano campionati dallo stesso ad una frequenza ben precisa, quindi, una volta ricevuti, non è necessario effettuare nessuna misura per calcolare l'intervallo di tempo trascorso fra 2 “report” (si supponga, ad esempio, di voler stimare la posizione integrando i segnali di accelerazione). Tale misura risulterebbe in ogni caso imprecisa a causa dei ritardi di comunicazioni dovuti al protocollo Bluetooth e al sistema operativo.

3.2.1.2 Ciclo principale

L'altro motivo che ci ha portati a impostare il flag `WIIUSE_CONTINUOUS` risiede nella struttura generale del software da noi sviluppato.

I moduli principali si attivano in sequenza a partire dal messaggio `MSG_TO_BRIAN` inviato da WiimExpert. Per questo motivo è ragionevole temporizzare lo stesso in modo preciso inviando messaggi ad intervalli regolari, indipendentemente dalla variazione o meno dello stato del Wiimote. D'altra parte temporizzare in modo arbitrario WiimExpert, indipendentemente dalla comunicazione con il Wiimote, avrebbe portato problemi alla gestione dello stack Bluetooth dei messaggi.

Il ciclo principale di WiimExpert risulta strutturato secondo il seguente schema:

```
se (ricevuto messaggio relativo alla posizione del robot) {
      memorizza la nuova posizione del robot
}
se (wiiuse_poll(wm, 1)==FALSE){
      return //nessun nuovo “report” → non è ancora trascoro 1/100 di secondo
```

```
}altrimenti{
      se (modalità == MANUALE){
            invia comandi direttamente a MotorExpert
      }altrimenti{
            esegui lo schema di gioco //vedi sezione 3.2.1
            calcola i valori di gioco
            invia messaggio a BrianExpert con i valori di gioco
      }
}
```

Le uniche istruzioni che vengono eseguite indipendentemente dal Wiimote sono quelle relative all'acquisizione del messaggio di posizione. Questo messaggio viene inviato dai moduli che calcolano la posizione. E' importante sottolineare che l'istruzione per la ricezione dei messaggi di tipo `MSG_POSITION` è eseguita in modo non bloccante, quindi, l'esecuzione di WiimExpert, nel caso non venga ricevuto nessun nuovo messaggio, prosegue. In questo caso, come posizione del robot, verrà utilizzata l'ultima ricevuta.

E' stata implementata anche una modalità di controllo manuale del robot, utile nella fase di test e messa a punto del sistema, tramite la quale è possibile controllare il robot, tramite la croce direzionale presente sul Wiimote, inviando direttamente i comandi al modulo MotorExpert.

Come è evidenziato anche dallo schema di gioco, il modo in cui effettivamente si deve muovere il robot non è gestito da WiimExpert, bensì da BrianExpert. WiimExpert si limita ad inviare a quest'ultimo il valore delle varie variabili (come, ad esempio, la distanza e l'angolo fra il robot e l'obiettivo, piuttosto che la posizione dove il Wiimote sta puntando) che utilizzerà per decidere il comportamento da attuare.

3.2.2 – BrianExpert

Il modulo BrianExpert si occupa di:

- ricevere i messaggi provenienti da WiimExpert (`MSG_TO_BRIAN`)
- effettuare il parser dei messaggi (tramite un parser creato con FLEX)
- stabilire i comandi da inviare ai motori tramite il software Mr. Brian (vedi sezione 2.4)
- inviare questi comandi in un messaggio al modulo MotorExpert (`MSG_TO_MOTION`)

I messaggi del tipo `MSG_TO_BRIAN` seguono la sintassi XML descritta nella 2.5.2, inoltre, per ogni variabile reale inviata è presente il seguente campo:

```
<A>   VALORE      AFFIDABILITA'     </A>
```

Il periodo di BrianExpert è stato impostato a 0, ma l'istruzione per la ricezione dei messaggi di tipo `MSG_TO_BRIAN` è eseguita in modo bloccante. Quindi BrianExpert verrà eseguito solamente in risposta ad un messaggio `MSG_TO_BRIAN`, inviato da WiimExpert.

I comportamenti impostati in Mr. Brian saranno analizzati nel dettaglio nella sezione 3.3.

Abbiamo inoltre aggiunto al modulo la possibilità di scrivere, istante per istante, i seguenti valori in un file di log:

- le variabili (reali) di ingresso
- i valori di CANDO e WANT per ogni comportamento
- i valori (reali) di uscita

I valori così registrati possono essere visualizzati in tempo reale tramite GnuPlot o Matlab (vedi sezione 4).

3.2.3 – MotorExpert

Il modulo MotorExpert si occupa di:

- ricevere i messaggi provenienti da BrianExpert (`MSG_TO_MOTION`)
- effettuare il parser dei messaggi (tramite un parser creato con FLEX)
- stabilire i comandi da inviare alla scheda di controllo
- leggere dalla scheda di controllo i valori degli encoder ed inviare un messaggio (`MSG_FROM_MOTION`) con i loro valori

Il periodo di MotorExpert è stato impostato a 0, ma l'istruzione per la ricezione dei messaggi di tipo `MSG_TO_MOTION` è eseguita in modo bloccante. Quindi MotorExpert verrà eseguito solamente in risposta ad un messaggio `MSG_TO_MOTION`, inviato da BrianExpert (in caso di controllo automatico) o direttamente da WiimExpert (in caso di controllo manuale).

Inoltre, per evitare di inviare troppi comandi alla scheda di controllo e saturare la porta seriale usata per la comunicazione, i comandi ad essa sono inviati solamente 5 volte ogni secondo.

I messaggi del tipo `MSG_TO_MOTION` e `MSG_FROM_MOTION` seguono la sintassi XML descritta nella sezione 2.5.2.

I messaggi `MSG_TO_MOTION`, nella nostra implementazione, contengono 2 campi ulteriori:

```
<D>   TanSpeed    VALORE      </D>
<D>   RotSpeed    VALORE      </D>
```

mentre nei messaggi di tipo `MSG_FROM_MOTION` utilizziamo i seguenti campi aggiuntivi:

```
<D>   movement    VALORE      </D>
<D>   orientation VALORE      </D>
```

Dai valori `TanSpeed` e `RotSpeed` sono calcolati i valori di velocità delle 2 ruote da inviare alla scheda di controllo, tramite la seguente formula:

```
Velocità_Motore_Dx = TanSpeed - RotSpeed
Velocità_Motore_Sx = TanSpeed + RotSpeed
```

Per calcolare l'odometria viene letto il valore di Tick dall'ultima lettura per ogni motore ed applicata la seguente formula:

```
mm_Sx = Tick_Motore_Sx * Coefficiente_Sx
mm_Dx = Tick_Motore_Dx * Coefficiente_Dx
movement = (mm_Sx+mm_Dx) / 2
orientation = ( mm_Sx - mm_Dx ) / distanza_ruote
```

dove `Coefficiente_Dx` e `Coefficiente_Sx` sono il numero di Tick corrispondenti ad uno spostamento di un millimetro.

Il valore `movement` rappresenta la variazione di posizione lungo l'orientamento attuale e `orientation` è la variazione nell'orientamento del robot (in radianti).

3.2.4 – PositionDummyExpert

Questo modulo calcola la posizione del robot direttamente dai valori inviati da BrianExpert ai motori (`MSG_TO_MOTION`). Ricevendo la velocità di rotazione e quella tangenziale desiderata del robot ne aggiorna la sua posizione, che invia tramite un messaggio di tipo `MSG_POSITION`.

Questo modulo è stato utilizzato solamente durante la fase di sviluppo del sistema per poter verificare i comportamenti, anche senza dover utilizzare il robot.

3.2.5 – VisionExpert

Il modulo VisionExpert determina la posizione del robot in base alla immagini provenienti dalla telecamera, utilizzando, come riferimento, marker posizionati sul soffitto (vedi sezione 2.6). La posizione restituita è nel piano, cioè espressa in termini di coordinate x ed y, e rotazione α. Il modulo è stato solo leggermente modificato rispetto a quello già sviluppato per il progetto Lurch.

Le nostre modifiche sono state le seguenti:

- Le coordinate restituite dal sistema di analisi dei frame sono state scalate e traslate, in un sistema più adatto al gioco da noi sviluppato.
- E' stato aggiunto un filtro a media mobile che restituisce la media degli ultimi 7 valori rilevati. La posizione è ritenuta valida se almeno in un frame, degli ultimi 7 analizzati, è stato rilevato un marker. La media è pesata in modo tale da dare maggiore valore agli ultimi dati rilevati.

Il periodo del modulo è stato impostato con un periodo di 100ms, ed invia quindi 10 messaggi al secondo di tipo `MSG_POSITION` contenenti la posizione del robot. Nel caso che la posizione rilevata non sia valida viene inviato un particolare messaggio, e in questo caso il modulo WiimExpert interrompe, per motivi di sicurezza, ogni movimento automatico del robot. Il modulo può essere impostato anche per inviare messaggi di tipo `MSG_POSITION_P` invece che `MSG_POSITION`. In questo caso i messaggi saranno ricevuti dal modulo PositionExpert, invece che da WiimExpert (vedi sezione 3.2.6).

3.2.6 – PositionExpert

Avendo disponibili sia le informazioni relative all'odometria che quelle di posizione assoluta provenienti da VisionExpert, abbiamo pensato a creare un modulo che integrasse le 2 informazioni. Infatti odometria e visione sono metodi complementari di localizzazione e il loro utilizzo combinato può portare a posizioni più precise.

Nella tabella seguente è proposto un confronto delle 2 tecniche.

ODOMETRIA	VISIONE
Informazione differenziale, istante per istante ho la velocità di spostamento	Informazione assoluta, istante per istante ho la posizione
Precisa (per piccoli spostamenti)	Poco precisa
Errore crescente	Errore costante
Informazioni odometriche sempre disponibili	Informazioni non sempre disponibili (marker al di fuori dal campo di visione o coperti, errata illuminazione, ...)

Lo schema di funzionamento complessivo del modulo PositionExpert è il seguente:

leggi messaggio inviato da VisionExpert

se (la posizione inviata da VisionExpert è valida)**{**

    invia la nuova posizione a WiimExpert

}altrimenti{

    leggi il messaggio inviato da MotorExpert

    se (l'ultima posizione valida ricevuta da VisionExpert è stata da meno di 10s)**{**

        stima la nuova posizione da i dati di odometria

        invia la nuova posizione a WiimExpert

    }altrimenti{

        invia messaggio di posizione non valida a WiimExpert

    }

}

La posizione è stimata tramite i dati di odometria con la seguente formula:

```
t = tempo trascorso dall'ultimo messaggio di odometria
nuovo_angolo = vecchio_angolo + velocità_rotazione * t
nuovo_x = vecchio_x + ( velocità_tangenziale * cos(nuovo_angolo) * t)
nuovo_y = vecchio_y + ( velocità_tangenziale * sin(nuovo_angolo) * t)
```

Abbiamo, inoltre, ritenuto opportuno considerare non valida la posizione, dopo 10 secondi che non vengono ricevuti messaggi di posizione valida da VisionExpert, in quanto, dopo questo intervallo di

tempo, l'errore accumulato dall'odometria risulta considerevole.

3.2.7 – LogExpert

Abbiamo creato un modulo il cui scopo principale è quello di visualizzare la posizione del robot e i dati provenienti dal sensore ad infrarossi. Questo ci è servito principalmente nella fase di sviluppo e messa a punto del sistema, in quanto ci ha permesso di sviluppare i comportamenti senza necessariamente verificarli con il robot. Inoltre ci ha permesso di verificare e calibrare il sistema di posizionamento.

Il modulo crea una finestra grafica utilizzando le librerie OpenGL.

Nella parte superiore della finestra appaiono i dati rilevati dal sensore ad infrarossi (vedi sezione 2.2.3.3), in particolare:

- i triangoli rossi rappresentano la posizione rilevata dei LED
- i triangoli gialli rappresentano la posizione degli stessi a cui è stato tolto l'effetto della rotazione
- il triangolo bianco rappresenta il punto effettivamente puntato dal Wiimote

Il cerchio interno (azzurro) rappresenta la zona entro la quale, nel caso venga premuto il pulsante B, il robot è considerato colpito.

Nella parte inferiore della finestra sono rappresentate:

- la posizione del robot ed il suo orientamento (freccia rossa)
- il campo di gioco (circonferenza azzurra)
- la posizione che il robot deve raggiungere per totalizzare un punto (circonferenza nera)
- l'area in cui il robot si posiziona per iniziare la partita o dopo che è stato totalizzato un punto (circonferenza bianca)

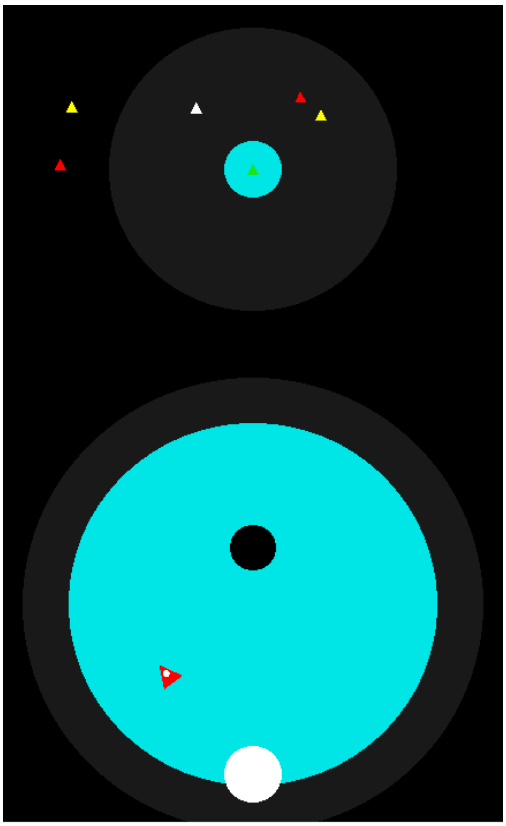


Figura 3.2: Esempio di finestra disegnata da LogExpert

3.3 – Configurazione comportamenti (Mr. Brian)

Di seguito illustreremo come abbiamo configurato Mr. Brian per ottenere il comportamento complessivo del robot da noi desiderato. Si rimanda all'appendice B per un elenco analitico della configurazione da noi usata.

3.3.1 – Dati in ingresso

Mr. Brian riceve in ingresso diversi valori dal modulo WiimExpert e in base ad essi

DtoGoal: distanza fra il robot e il goal

AtoGoal: angolo fra il robot e il goal

IRCenter: distanza fra il centro del bersaglio e il punto puntato dal Wiimote, questo valore è positivo se il Wiimote punta una zona a destra del bersaglio e negativo se punta una zona a sinistra

Arobot: orientamento assoluto del robot

DtoHome: distanza fra il robot e la home, ovvero l'area in cui posizionarsi all'inizio del gioco, o quando viene totalizzato un punto

AtoHome: angolo fra il robot e la home

Hit: indica se è stato totalizzato un punto e il robot deve tornare nella home

HomeRot: indica se il robot deve orientarsi, all'interno della home, in posizione di inizio gioco

DtoCenter: distanza fra il robot ed il centro del campo di gioco

AtoCenter: angolo fra il robot e il centro del campo di gioco

WiiMotePitch: indica l'inclinazione del Wiimote lungo l'angolo di pitch

GAcc: indica l'accelerazione complessiva a cui è sottoposto il Wiimote

Random: valore che alterna in modo ciclico fra -1 e 1

ProposedTanSpeed: velocità tangenziale proposta dai comportamenti di livello 1

3.3.2 – Comportamenti

Per il funzionamento di Robowii, abbiamo sviluppato 6 comportamenti diversi:

AligntoGoal: si occupa di allineare il robot con il Goal, cioè l'area che deve raggiungere per totalizzare un punto (livello 1)

GotoGoal: si occupa di muovere il robot verso il Goal, nel caso sia allineato con esso (livello 1)

Snaking: fa assumere al robot un comportamento più cauto, nel caso sia probabile che l'umano stia tentando di colpirlo (livello 2)

EscapeIR: si occupa di muovere il robot in modo tale da evitare il puntamento col Wiimote da parte dell'umano (livello 3)

RobotHit: se il robot o l'umano totalizzano un punto, questo comportamento fa tornare il robot nella sua posizione iniziale (livello 4)

StayInArea: evita che il robot esca dall'area di gioco (livello 5)

Di seguito li analizzeremo uno per uno nel dettaglio.

3.3.2.1 – AlignToGoal e GotoGoal

Questi comportamenti sono situati al livello più basso della scala gerarchica di Mr. Brian. Infatti abbiamo deciso che il robot deve tentare di totalizzare un punto solo quando non ha nessun'altro obiettivo. In particolare, evitare di essere colpito, è prioritario rispetto a raggiungere il Goal.

I due comportamenti agiscono contemporaneamente in modo sinergico, uno allineando il robot verso il Goal e l'altro facendolo spostare verso di esso.

Le condizioni di CANDO e WANT per AlignToGoal sono:

```
AlignToGoal CANDO:  (P GoalNotTooClose);
AlignToGoal WANT: (P GoalNotAligned);
```

Queste condizioni impongono che il robot può allinearsi al Goal se non è già allineato ad esso o se non è troppo vicino ad esso. Questa scelta serve ad evitare che il robot cerchi di allinearsi al Goal quando, in pratica, si trova già su di esso.

Le condizioni di CANDO e WANT per GotoGoal sono:

```
GotoGoal CANDO: (AND (P GoalNotTooClose) (P GoalAligned));
GotoGoal WANT: (P GoalNotTooClose);
```

Queste condizioni fanno si che il robot si avvicini al Goal solo se non è già su di esso. Inoltre evitano che il robot tenti di muoversi verso il Goal se non è orientato in modo corretto.

Il comportamento AlignToGoal è composto dalle seguenti regole:

```
(GoalANearL) => (RotSpeed LEFT);
(GoalANearR) => (RotSpeed RIGHT);
(GoalAFarL) => (RotSpeed FAST_LEFT);
(GoalAFarR) => (RotSpeed FAST_RIGHT);
```

Queste regole fanno allineare il robot al Goal più velocemente se l'angolo di rotazione da compiere è maggiore. Inoltre abbiamo deciso che il robot, potendo muoversi sia in avanti che all'indietro, si può allineare al Goal in entrambi i modi. Questo è stato ottenuto definendo i predicati `GoalANearL`, `GoalANearL`, `GoalAFarL`, `GoalAFarR`, come visualizzato in figura 3.3:

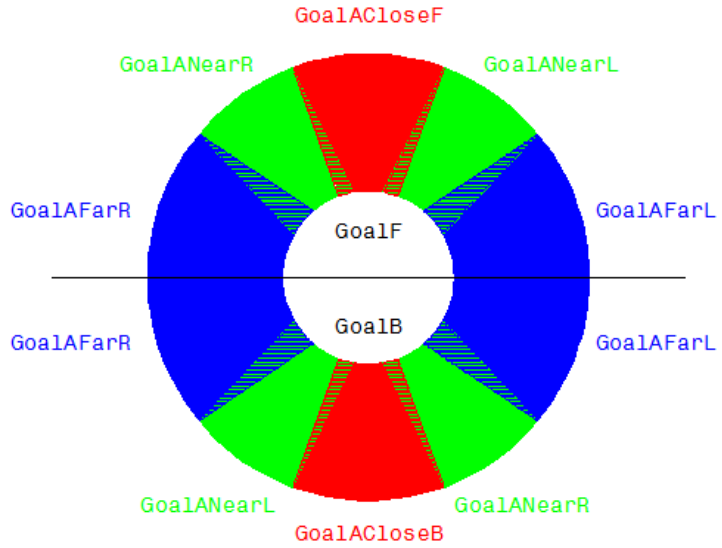


Figura 3.3: Rappresentazione grafica del valore dei predicati fuzzy[1] principali utilizzati dai comportamenti AlignToGoal e GotoGoal.

Il comportamento GotoGoal è composto dalle seguenti regole:

```
(AND (GoalF) (GoalDNear)) => (TanSpeed FORWARD);
(AND (GoalF) (GoalDFar)) => (TanSpeed FAST_FORWARD);
(AND (GoalB) (GoalDNear)) => (TanSpeed BACKWARD);
(AND (GoalB) (GoalDFar)) => (TanSpeed FAST_BACKWARD);
```

anche in questo caso il robot può raggiungere il Goal sia muovendosi in avanti (in questo caso sarà vero il predicato `GoalF`) che all'indietro (in questo caso sarà vero il predicato `GoalB`).

3.3.2.2 – Snaking

Per rendere il comportamento complessivo del robot meno prevedibile, abbiamo implementato questo comportamento che lo fa rallentare e spostare a zigzag se è probabile che l'umano stia cercando di mirare al robot. Questo è ottenuto analizzando l'angolo di pitch del Wiimote, che deve essere leggermente puntato verso il basso e l'accelerazione complessiva dello stesso.

Le condizioni di CANDO e WANT sono:

```
CANDO: Snaking = (OR (P GAccExist) (P Pointing));
WANT: Snaking = (OR (P GAccExist) (P Pointing));
```

e fanno si che questo comportamento sia attivo solo quando è presente un'accelerazione sufficientemente intensa sul Wiimote o l'angolo di pitch dello stesso sia leggermente verso il basso.

Le regole che compongono il comportamento sono le seguenti:

```
(GAccWeakR) => (RotSpeed RIGHT);
(GAccStrongR) => (RotSpeed FAST_RIGHT);
(GAccWeakL) => (RotSpeed LEFT);
```

```
(GAccStrongL) => (RotSpeed FAST_LEFT);
(AND (Pointing) (RandomPlus)) => (RotSpeed FAST_LEFT);
(AND (Pointing) (RandomMinus)) => (RotSpeed FAST_RIGHT);

(propTanSpeedFF) => (TanSpeed FORWARD);
(propTanSpeedF) => (TanSpeed SLOW_FORWARD);
(propTanSpeedB) => (TanSpeed BACKWARD);
(propTanSpeedFB) => (TanSpeed SLOW_BACKWARD);
```

I predicati del tipo `GAcc` e `Random` cambiano valore ciclicamente ogni 2 secondi, permettendo un andamento a ZigZag. I predicati del tipo `propTanSpeed` corrispondono alla velocità tangenziale proposta dal livello 1 che viene diminuita.

3.3.2.3 – EscapeIR

Il comportamento EscapeIR serve per far fuggire il robot nel caso sia puntato dal Wiimote.

Le regole che lo compongono sono le seguenti:

```
(IRDirClose) => (&DEL.RotSpeed ANY)(&DEL.TanSpeed ANY);
(IRDirClose) => (TanSpeed FAST_BACKWARD);
(IRDirNear) => (TanSpeed BACKWARD);
(IRDirCloseR) => (RotSpeed FAST_RIGHT);
(IRDirCloseL) => (RotSpeed FAST_LEFT);
(IRDirNearR) => (RotSpeed RIGHT);
(IRDirNearL) => (RotSpeed LEFT);
```

`IRDirClose` indica che il robot è puntato in modo preciso (può essere colpito se viene premuto B), `IRDirNear` indica che il robot è puntato, ma in modo non sufficientemente preciso per essere colpito.

Le regole sopra enunciate fanno si che:

- se il robot è puntato in maniera precisa annulla tutti gli spostamenti suggeriti dai livelli inferiori (livelli 1 e 2) e va velocemente indietro
- altrimenti si muove indietro, ma più lentamente
- in ogni caso ruota dalla parte opposta rispetto a quella dove è puntato

3.3.2.4– RobotHit

Questo comportamento ha lo scopo, nel caso che venga totalizzato un punto dal robot o dall'umano, di fare tornare il robot nella posizione iniziale di Home orientato nel verso giusto.

Per fare questo, quando viene totalizzato un punto, il modulo WiimExpert invia il dato Hit con valore 1 che fa si che il comportamento RobotHit venga attivato. Quando il robot ha raggiunto l'area di Home, il modulo WiimExpert invia il dato HomeRot con valore 1, che fa si che il robot si allinei correttamente. A questo punto il gioco riprende e i dati Hit ed HomeRot non sono più inviati.

Quando il comportamento è attivo vengono annullati tutti gli spostamenti proposti dai livelli inferiori.

3.3.2.5 – StayInArea

Questo comportamento evita che il robot esca dall'area di gioco.

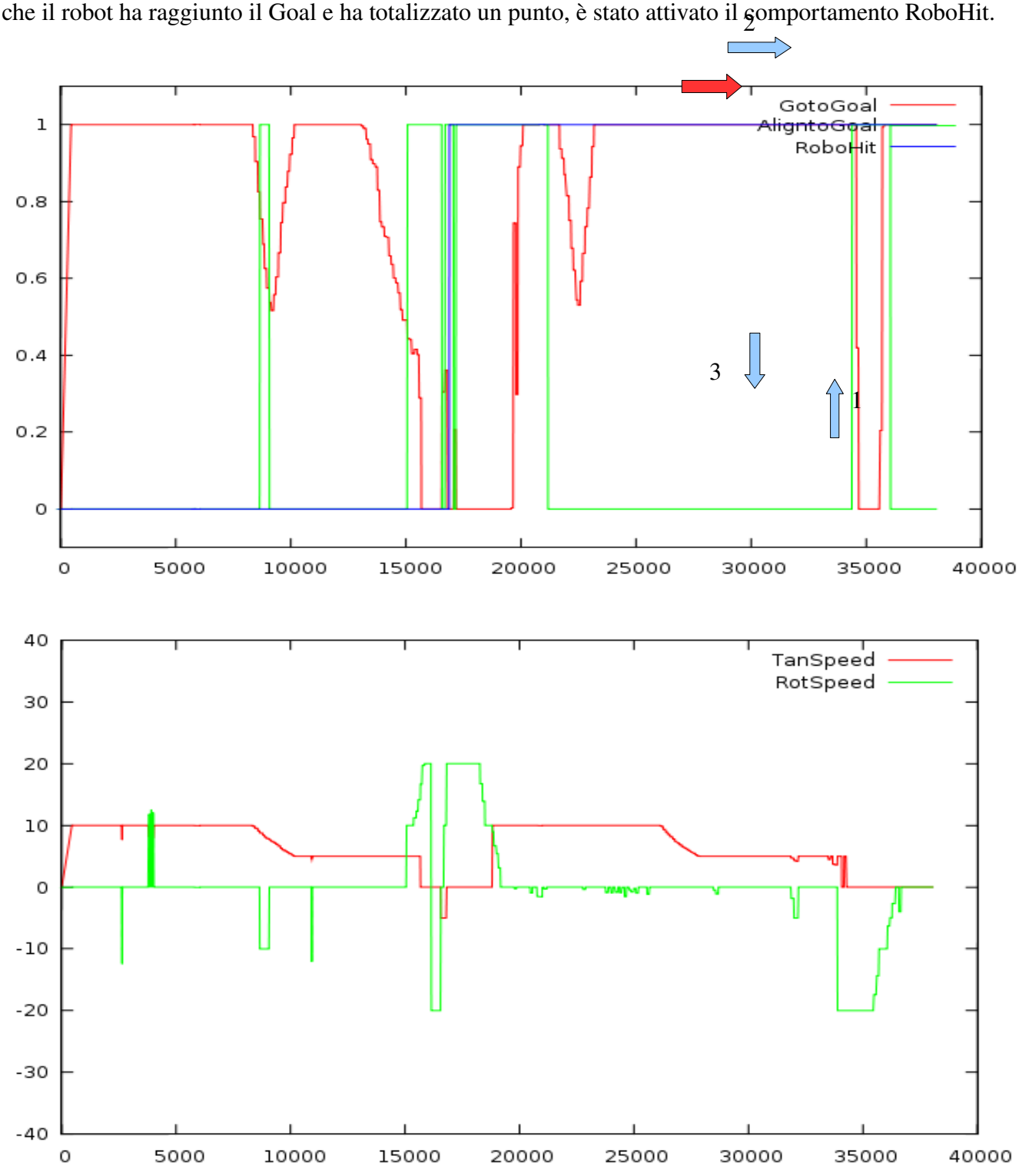
Il comportamento agisce in modo diverso nel caso che il robot si trovi all'estremo dell'area di gioco o al di fuori di essa. Nel primo caso viene annullato ogni spostamento tangenziale proposto dai livelli precedenti e imposto uno spostamento in avanti o all'indietro (a secondo dell'orientamento del robot) per far si che esso si diriga verso il centro dell'area di gioco. Nel secondo caso sono annullati anche i movimenti di rotazione, il robot viene prima fatto allineare con il centro dell'area di gioco e poi mosso verso di essa.

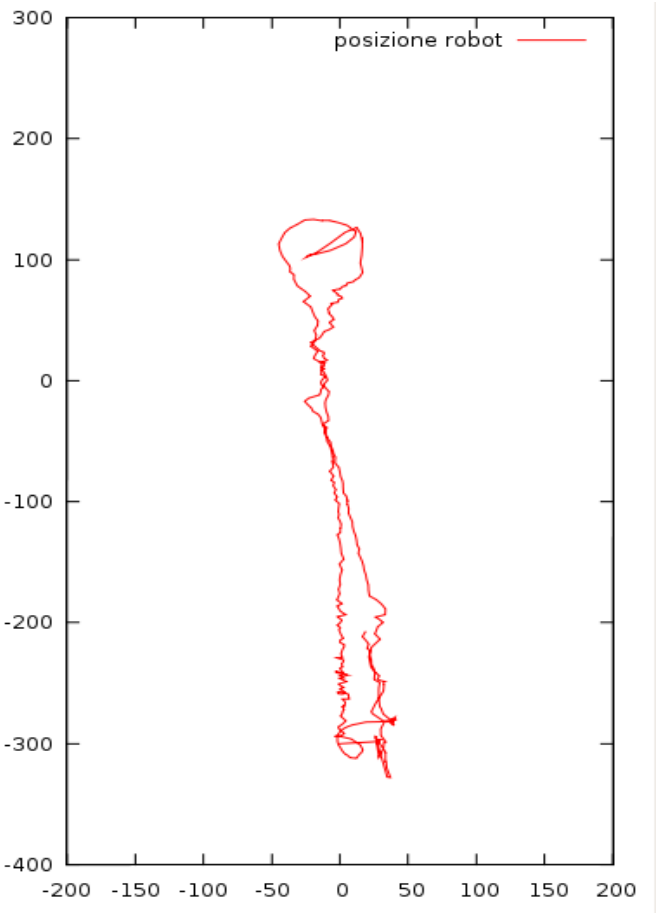
4 – RISULTATI

Di seguito illustreremo varie sessioni di gioco. Di esse abbiamo generato i grafici rappresentanti la posizione del robot, i livelli di attivazione dei comportamenti principali e i comandi inviati alla scheda di controllo. Per quanto riguarda i grafici dei livelli di attivazione dei comportamenti abbiamo disegnato il minore fra il valore di CANDO e quello di WANT, il che equivale a mettere in AND le 2 condizioni. Il tempo sui grafici è espresso in millisecondi, mentre per far riferimento ad istanti o intervalli temporali si indicheranno i secondi fra parentesi tonde.

4.1 – AlignToGoal e GotoGoal

In questa prima sessione di gioco abbiamo attivato il robot, mentre il giocatore rimaneva fermo, quindi, gli unici comportamenti attivi sono stati inizialmente AlignToGoal e GotoGoal. Successivamente, dopo che il robot ha raggiunto il Goal e ha totalizzato un punto, è stato attivato il comportamento RoboHit.





Dal grafico della posizione del robot possiamo vedere che esso si è mosso seguendo una traiettoria, in larga misura, rettilinea, sia durante la fase di avvicinamento al Goal sia durante quella di ritorno alla Home. Dagli altri grafici si può rilevare che (fra parentesi l'intervallo di tempo, in secondi, in cui l'evento avviene):

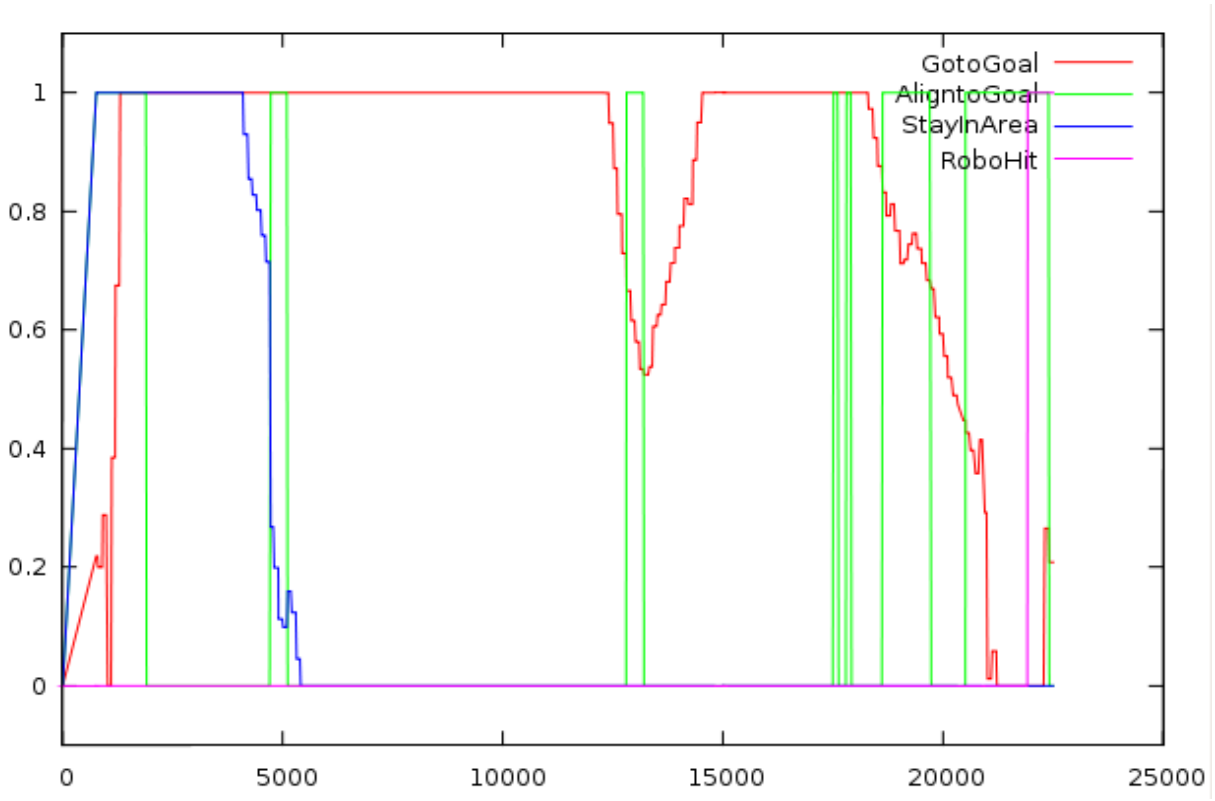
- (7) il robot effettua una correzione della traiettoria, ruotando leggermente.
- (8-15) il robot rallenta, trovandosi nelle vicinanze del Goal.
- (15-17) effettua un'altra correzione della traiettoria a causa del comportamento (AligntoGoal).
- (17) il robot realizza un punto. A questo punto prima che il gioco riprenda, il robot deve ritornare alla Home ed il comportamento RoboHit è attivo. Essendo questo comportamento attivo, i livelli di attivazione di AligntoGoal e GotoGoal non sono più significativi.
- (17-19) il robot si allinea verso la Home.
- (19-34) il robot raggiunge la Home
- (35-37) il robot si allinea verso il Goal

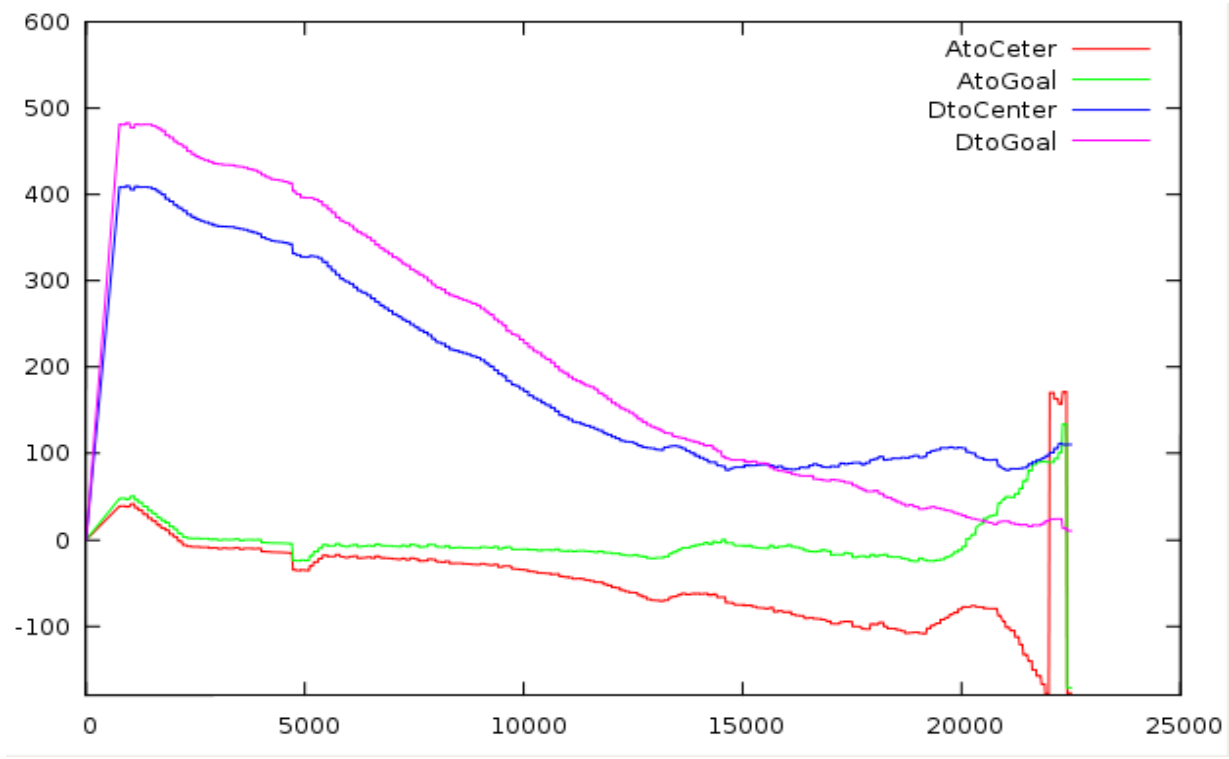
A questo punto il gioco riprende.

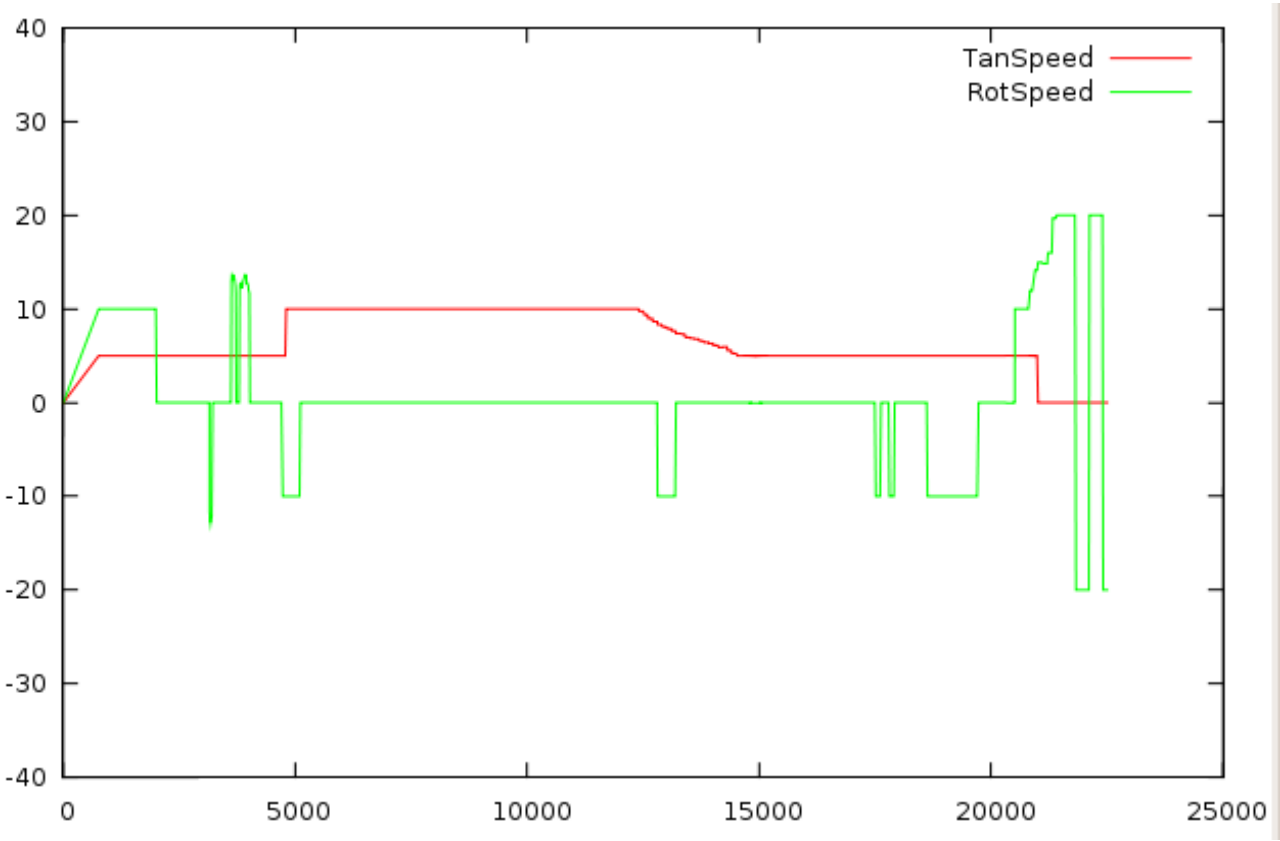
Possiamo notare come il sistema di localizzazione, calcoli la posizione del robot in maniera abbastanza precisa, anche se a volte commette degli errori. Ad esempio, nel punto indicato dalla freccia rossa viene commesso un errore nel calcolo della posizione.

4.2 – StayInArea

In questa sessione abbiamo verificato il funzionamento del comportamento StayInArea, che fa si che il robot non si allontani troppo dall'area di gioco e nel caso in cui questo accada (ad esempio per un temporaneo errore nel calcolo della posizione) fa dirigere il robot verso il centro.





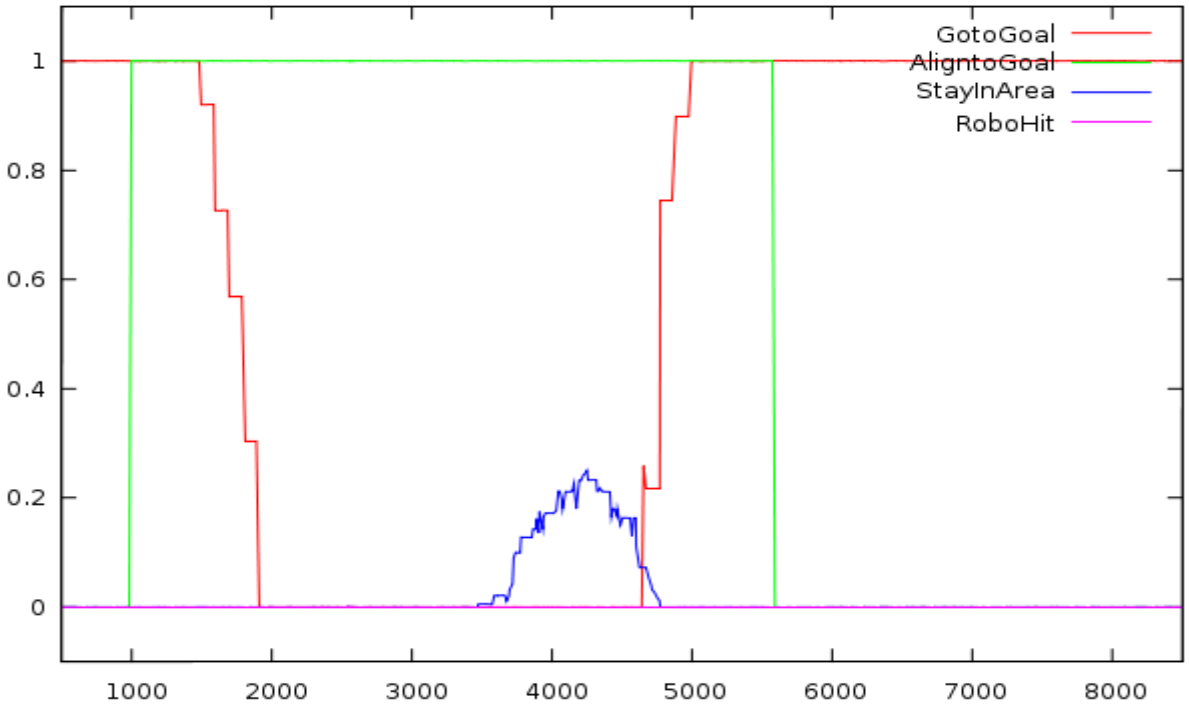


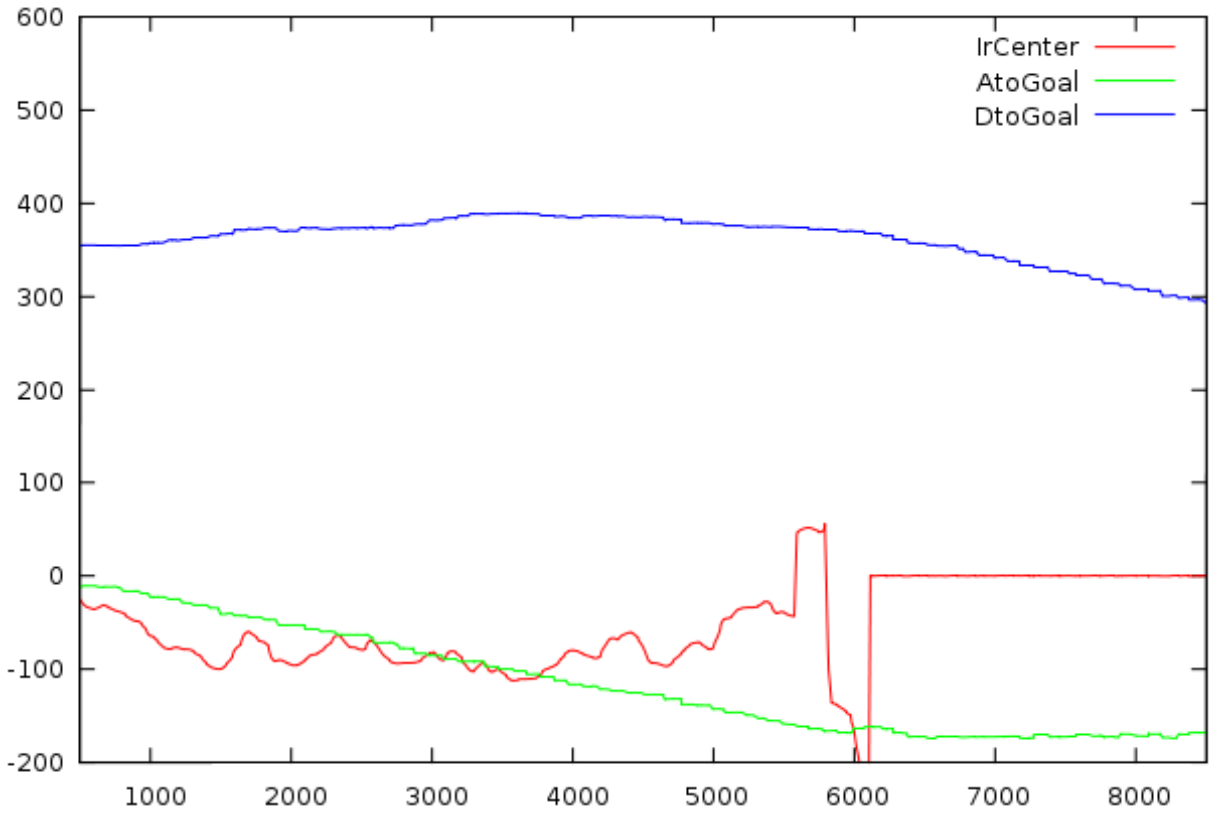
Dai grafici possiamo vedere che (escludendo il primo secondo, in cui il robot è in controllo manuale):

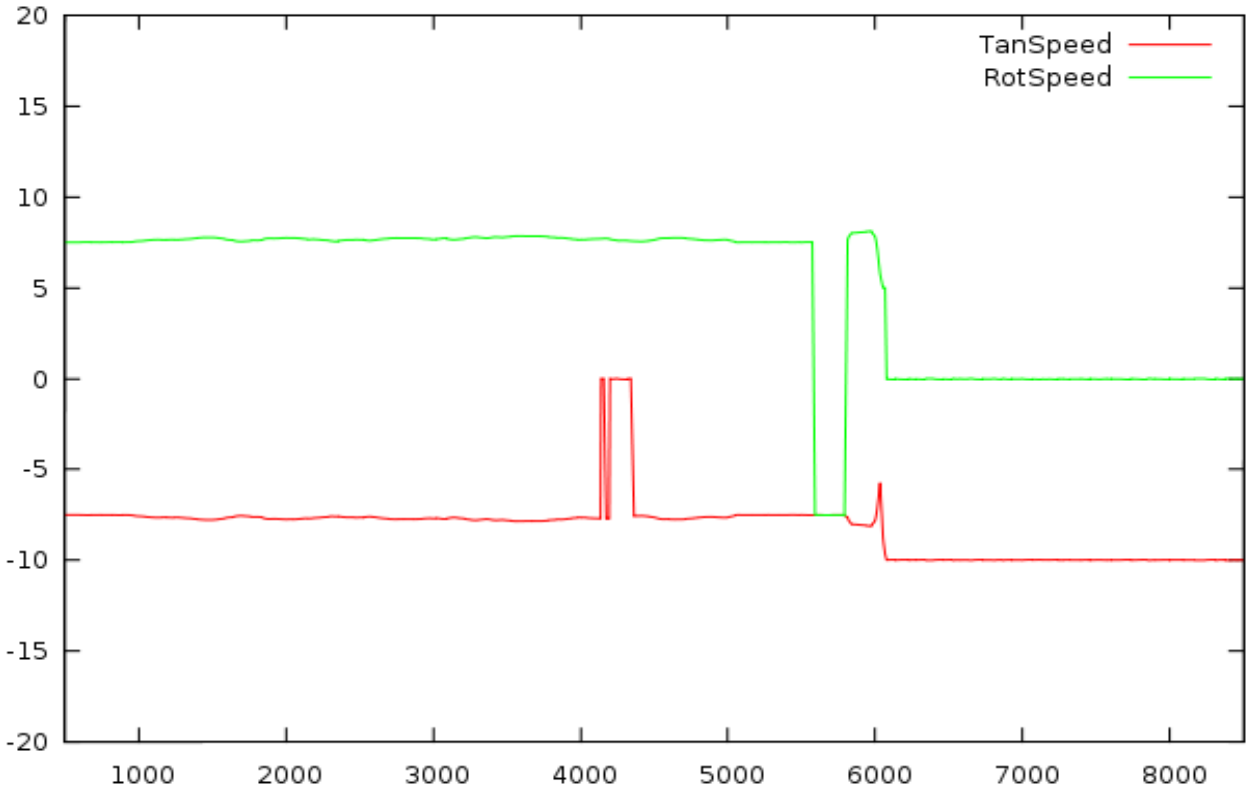
- (1-2) Il comportamento StayInArea è attivo, il robot inizia ad allinearsi con il centro (AtoCenter diminuisce).
- (2-5) Il robot incomincia a muoversi lentamente verso il centro (DtoCenter diminuisce) e continua ad allinearsi ad esso (AtoCenter tende a 0).
- (5) Non essendo più attivo il comportamento StayInArea il robot effettua una piccola correzione di orientamento che lo porta ad allinearsi verso il Goal, invece che verso il centro.
- (5-13) Il robot è già sostanzialmente allineato verso il Goal, quindi si dirige verso il Goal.
- (13) Il robot si allinea in modo migliore verso il Goal, si può notare che il livello di GotoGoal scende intorno al secondo 13, appunto a causa del cattivo allineamento fra il robot ed il Goal.
- (13-18) Il robot prosegue verso il Goal.
- (18-22) Trovandosi già vicino al Goal, lo spostamento del robot verso di esso rallenta. L'angolazione viene corretta più volte.
- (23) Il robot ha raggiunto il Goal e ha mantenuto la posizione per più di 3 secondi, per questo realizza un punto.

4.3 – EscapeIR

In questa sessione abbiamo verificato il comportamento di fuga del robot dal giocatore, quando quest'ultimo tenta di colpirlo con il Wiimote.







Dai grafici si vede che:

- (0.5-4) Il robot è puntato col Wiimote da sinistra (IrCenter < 0), per sfuggire al giocatore ruota verso sinistra (RotSpeed < 0) e si muove all'indietro leggermente (TanSpeed = -7). L'angolo fra il robot ed il Goal aumenta, non essendo più il robot allineato al Goal il livello di attivazione del comportamento GotoGoal va a 0.
- (4-4.5) Il robot, muovendosi all'indietro, si è allontanato dal centro del campo di gioco. Per questo motivo il comportamento StayInArea si è attivato (livello di attivazione > 0.2). Questo comportamento evita che il robot esca dall'area di gioco e per fare questo impone che la velocità tangenziale sia uguale a 0.
- (4.5-5.7) Il robot continua a girare ed indietreggiare per evitare il Wiimote.
- (5.7) Il giocatore punta il Wiimote dalla direzione opposta, la velocità angolare si inverte.
- (5.7-8.5) Il giocatore non punta più il robot. L'angolo fra il robot e il Goal è di circa 170 gradi, quindi il robot si trova già allineato al Goal, anche se per raggiungerlo deve muoversi all'indietro. Il moto del robot proseguo quindi con RotSpeed = 0 e TanSpeed < 0.

5 – CONCLUSIONI

Durante il nostro lavoro abbiamo sviluppato un sistema di gioco basato sull'interazione del Wiimote con un robot a ruote, abbiamo integrato diversi componenti (comunicazione con il Wiimote, comunicazioni con il robot, localizzazione, odometria) utilizzando DCDT e creando un unico ambiente software in cui essi collaborano.

A partire dal nostro lavoro sono possibili vari sviluppi.

5.1 – Sviluppi del sistema di gioco

Per prima cosa è possibile creare vari livelli di difficoltà del gioco, rendendo, ad esempio, più o meno reattivo il comportamento del robot. Ad esempio si può pensare a variare la velocità di rotazioni che il robot ha quando cerca di evitare di essere puntato, oppure di rendere più o meno veloce il suo dirigersi verso il Goal.

Si potrebbe rendere il robot colpibile in più punti, ad esempio sia anteriormente che posteriormente, per fare questo sarebbe necessario collocare ulteriori LED su di esso, però bisognerebbe fare in modo che il robot potesse capire verso quale bersaglio il giocatore stia puntando. Questo potrebbe essere reso possibile posizionando i LED secondo configurazioni ben definite e diverse per ogni bersaglio.

Si può anche pensare ad un'estensione della nostra applicazione nella quale più persone collaborano per colpire il robot (compito facilmente realizzabile in quanto le librerie da noi utilizzate permettono di gestire più Wiimote contemporaneamente), oppure si potrebbe far si che il robot tenti di colpire il giocatore, che in questo caso dovrebbe indossare un qualche dispositivo con LED ad infrarossi.

L'interattività del gioco potrebbe essere migliorata aggiungendo suoni provenienti dal Wiimote, questo è tecnicamente possibile in quanto il Wiimote presenta uno speaker al suo interno, ma il suo funzionamento non è stato ancora totalmente implementato in nessuna libreria disponibile.

Un'altra possibile miglioria al gioco è quella di rendere il robot in grado di capire gli spostamenti e il tipo di movimento che il giocatore sta facendo. Per questi scopi si potrebbero usare i dati provenienti dall'accelerometro, dati che, però, risultano altamente rumorosi e comunque incompleti per poter determinare l'orientamento del Wiimote nello spazio, soprattutto se in movimento. Nonostante queste difficoltà tecniche esistono librerie, appositamente studiate per riconoscere i movimenti effettuati impugnando il Wiimote, che si basano su tecniche di apprendimento e riconoscimento. Tramite di esse sarebbe possibile capire se il giocatore si sta muovendo o semplicemente sta ruotando, tenendolo fermo, il Wiimote e quindi stimare la sua posizione, in modo da evitarlo.

La stima della posizione dell'umano potrebbe essere fatta anche utilizzando tecniche più sofisticate basate sui LED come quella esposta nell'appendice A.

5.2 – Sviluppi del sistema di localizzazione

Il sistema di localizzazione da noi utilizzato, basato su marker ed odometria, può essere migliorato in molti suoi aspetti. In primo luogo la stima dei parametri z, β e γ, fatta durante la fase di calibrazione per diminuire l'errore di calcolo della posizione del robot rispetto ai marker, è comunque non ottimale, in quanto presuppone che, una volta effettuata la calibrazione, la telecamera rimanga sempre esattamente nella stessa posizione relativa rispetto al robot. Si potrebbe pensare ad un metodo di calibrazione diverso, oppure all'utilizzo di più marker contemporaneamente, per la stima della posizione. Calcolando la posizione del robot rispetto a più marker si potrebbe trovare un dato più affidabile.

Attualmente i dati di odometria sono utilizzati solamente quando non è disponibile il dato di posizione in arrivo dal sistema di visione, invece, sarebbe più opportuno utilizzarli in ogni momento per correggere la posizione stimata dal sistema di visione tramite, ad esempio, filtro di Kalman.

APPENDICE A

Determinare la posizione e l'orientamento del Wiimote nello spazio non è possibile né utilizzando l'accelerometro né il sensore ad infrarossi e 2 LED, così come accade durante il suo normale utilizzo.

Abbiamo cercato di sviluppare un sistema di localizzazione del Wiimote in 6 gradi di libertà, sfruttando l'algoritmo di calcolo della posizione dei marker utilizzato dalla libreria ArToolKit. Infatti, al posto di un marker è possibile utilizzare 4 LED posizionati su uno stesso piano in modo da formare un quadrato.

Più precisamente, la funzione `ArGetTransMat` della libreria ArToolKit richiede in ingresso una struttura dati che definisce la posizione sul piano immagine del marker. Nel nostro caso, invece, riempiremo questa struttura dati con le coordinate dei LED rilevate dal sensore ad infrarossi del Wiimote.

In particolare `ArGetTransMat` richiede i seguenti dati:

- coordinate dei 4 spigoli del marker, da noi saranno sostituite con le coordinate dei LED rilevate dal sensore ad infrarossi (wm->ir.dot[i].rx wm->ir.dot[i].ry)
- equazioni (nella forma `ax+by+c=0`) delle rette passanti per gli spigoli del marker, nel nostro caso verranno calcolate le equazioni passanti per le coordinate dei LED rilevati
- coordinate del centro del marker, da noi sostituite con coordinate con: wm->ir.ax wm->ir.ay
- un parametro che indica quale dei 4 angoli si trova più in altro nell'immagine (e determina quindi l'orientamento del marker)
- un parametro che indica la lunghezza (reale) di un lato del marker, da noi sostituito con la distanza a cui abbiamo posizionato i LED

Nel nostro caso determinare quale dei 4 LED si trovi più in alto nell'immagine non è facile come nel caso di marker, dove, salvo simmetrie, ogni angolo è ben identificabile. Infatti, il Wiimote non implementa nessun meccanismo per poter distinguere un LED da un altro e l'ordine, in cui vengono restituiti dalle librerie, è arbitrario. Però, è possibile sfruttare l'informazione dell'accelerometro (angolo di roll) per capire qual è il LED che si trova più in alto (vedi figura 6.1).

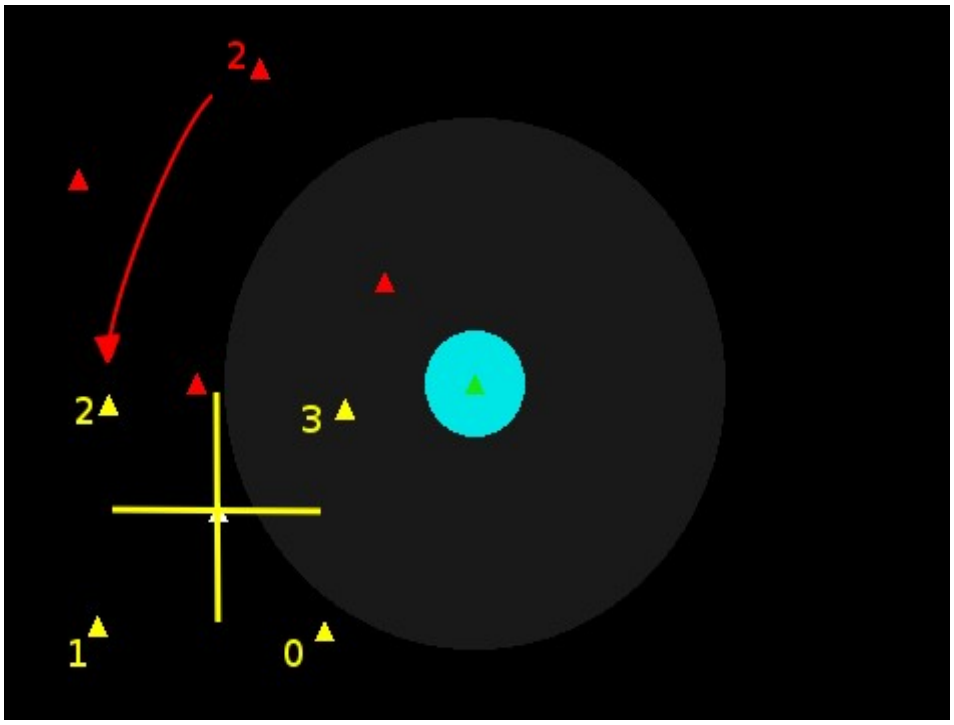


Figura 6.1: Per determinare qual è il LED posizionato più in alto è necessario eliminare l'effetto della rotazione roll. A questo punto è possibile numerare le coordinate dei punti (gialli) così ottenuti determinando la loro posizione rispetto al centro (bianco).

Invertendo la matrice di rototraslazione restituita da `ArGetTransMat`, si può ricavare la posizione del Wiimote rispetto al quadrato di LED. La misura è soggetta agli stessi errori ottenuti utilizzando questa tecnica per la localizzazione del robot. Si può però pensare di correggere la matrice inserendo in essa i valori di rotazione determinati dall'accelerometro prima di calcolarne l'inversa, in modo simile a come, nel meccanismo di localizzazione del robot, gli angoli β e γ venivano determinati durante la fase di calibrazione.

APPENDICE B

Di seguito elenchiamo il codice da noi usato per configurare Mr. Brian.

Variabili in ingresso

(DtoGoal DDISTANCE) (AtoGoal A2DISTANCE) (IRCenter IRDIR) (ARobot ADISTANCE) (DtoHome DDISTANCE) (AtoHome ADISTANCE) (HomeRot HIT) (Hit HIT)	(DtoCenter CDDISTANCE) (AtoCenter CADISTANCE) (WiiMotePitch PANGLE) (GAcc ACC) (Random RAN) (ProposedTanSpeed TAN2SPEED)

Insiemi fuzzy delle variabili in ingresso

```
(DDISTANCE
(TRA (CLOSE 0 0 30 50))
(TRA (NEAR 0 50 100 150))
(TOR (FAR 100 150))
)
(ADISTANCE
(TRA (CLOSE -20 -10 10 20))
(TRA (NEARL 5 10 50 60))
(TRA (NEARR -60 -50 -10 -5))
(TOR (FARL 40 50))
(TOL (FARR -50 -40))
)

(A2DISTANCE
(TRA (CLOSEF -10 -10 10 10))
(TRA (NEARLF 5 10 50 60))
(TRA (NEARRF -60 -50 -10 -5))
(TRA (FARLF 40 50 90 90))
(TRA (FARRF -90 -90 -50 -40))
(TRA (FARRB 90 90 130 140))
(TRA (FARLB -140 -130 -90 -90))
(TRA (NEARRB 120 130 177 178))
(TRA (NEARLB -178 -177 -130 -120))
(TOL (CLOSEBL -170 -170))
(TOR (CLOSEBR 170 170))
)
(IRDIR
(TRA (CLOSEL -100 -50 0 0))
(TRA (CLOSER 0 0 50 100))
(TRA (NEARL -300 -50 0 0))
(TRA (NEARR 0 0 50 300))
(TOL (FARL -640 -200))
(TOR (FARR 200 640))
(TRA (EXIST -640 -640 640 640))
)
(HIT
(SNG (V 1))
(SNG (F 0))
)
```

```
(CDDISTANCE
(TRA (NEAR 0 0 325 350))
(TOR (FAR 325 350))
(TOR (VERY_FAR 390 400))
)
(CADISTANCE
(TRA (CLOSE -30 -25 25 30))
(TRA (NEARL 0 0 60 70))
(TRA (NEARR -70 -60 0 0))
(TOR (FARL 50 60))
(TOL (FARR -60 -50))
(TRA (NEARF -85 -75 75 85))
(TOL (NEARBL -105 -95))
(TOR (NEARBR 95 105))
(TRA (FARML -110 -105 -75 -70))
(TRA (FARMR  70 75 105 110))
)
(PANGLE
(TRI (POINTING -5 30 70))
)
(ACC
(TRA (WEAK 0 0 25 30))
(TOR (STRONG 20 25))
(TOR (EXIST 13 13))
)
(RAN
(SNG (PLUS 1))
(SNG (MINUS -1))
)
(TAN2SPEED
(TOR (FAST_FORWARD 5 10))
(TRA (FORWARD 0 0 10 15))
(TRA (BACKWARD -15 -10 0 0))
(TOL (FAST_BACKWARD -10 -5))
)
```

Predicati

```
###distanza modulo, angolo dal Goal ###
GoalDClose = (D DtoGoal CLOSE);
GoalDNear = (D DtoGoal NEAR);
GoalDFar = (D DtoGoal FAR);


GoalACloseF = (D AtoGoal CLOSEF);
GoalANearLF = (D AtoGoal NEARLF);
GoalANearRF = (D AtoGoal NEARRF);
GoalAFarLF = (D AtoGoal FARLF);
GoalAFarRF = (D AtoGoal FARRF);
GoalACloseBR = (D AtoGoal CLOSEBR);
GoalACloseBL = (D AtoGoal CLOSEBL);
GoalACloseB    =    (OR    (P    GoalACloseBR)    (P
GoalACloseBL));
GoalANearLB = (D AtoGoal NEARLB);
GoalANearRB = (D AtoGoal NEARRB);
GoalAFarLB = (D AtoGoal FARLB);
GoalAFarRB = (D AtoGoal FARRB);


#altri predicati per il Goal
GoalNotTooClose = (OR (P GoalDNear)(P GoalDFar));


GoalAligned  =(NOT  (OR  (OR  (P  GoalAFarLF)  (P
GoalAFarRF)) (OR (P GoalAFarLB) (P GoalAFarRB))));
GoalNotAligned  =  (NOT  (OR  (P  GoalACloseF)  (P
GoalACloseB)));


GoalANearL = (OR (P GoalANearLB) (P GoalANearLF));
GoalANearR = (OR (P GoalANearRB) (P GoalANearRF));
GoalAFarR = (OR (P GoalAFarRB) (P GoalAFarRF));
GoalAFarL = (OR (P GoalAFarLB) (P GoalAFarLF));


GoalF   =   (OR   (OR   (P   GoalACloseF)   (OR   (P
GoalANearLF) (P GoalANearRF))) (OR (P GoalAFarLF)
(P GoalAFarRF)));
GoalB   =   (OR   (OR   (P   GoalACloseB)   (OR   (P
GoalANearLB) (P GoalANearRB))) (OR (P GoalAFarLB)
(P GoalAFarRB)));
################################################
```

```
###distanza modulo,angolo da Home###
HomeDClose = (D DtoHome CLOSE);
HomeDNear = (D DtoHome NEAR);
HomeDFar = (D DtoHome FAR);


HomeAClose = (D AtoHome CLOSE);
HomeANearL = (D AtoHome NEARL);
HomeANearR = (D AtoHome NEARR);
HomeANear = (OR (P HomeANearL)(P HomeANearR));
HomeAFarL = (D AtoHome FARL);
HomeAFarR = (D AtoHome FARR);


#angolo assoluto del robot
RobotAClose = (D ARobot CLOSE);
RobotANearL = (D ARobot NEARL);
RobotANearR = (D ARobot NEARR);
RobotAFarL = (D ARobot FARL);
RobotAFarR = (D ARobot FARR);
################################################


###IR###
#direzione
IRDirCloseR = (D IRCenter CLOSER);
IRDirCloseL = (D IRCenter CLOSEL);
IRDirNearR = (D IRCenter NEARR);
IRDirNearL = (D IRCenter NEARL);
IRDirFarR = (D IRCenter FARR);
IRDirFarL = (D IRCenter FARL);


IRDirClose = (OR (P IRDirCloseR) (P IRDirCloseL));
IRDirNear = (OR (P IRDirNearR) (P IRDirNearL));
IRExist = (D IRCenter EXIST);
################################################


###colpito (->goto Home)###
HitV = (D Hit V);
HomeRotV = (D HomeRot V);
################################################
```

```
###distanza modulo,angolo dal centro####
CenterDNear = (D DtoCenter NEAR);
CenterDFar = (D DtoCenter FAR);
CenterDVery_Far= (D DtoCenter VERY_FAR);
CenterAClose = (D AtoCenter CLOSE);
CenterANearL = (D AtoCenter NEARL);
CenterANearR = (D AtoCenter NEARR);
CenterAFarL = (D AtoCenter FARL);
CenterAFarR = (D AtoCenter FARR);

CenterANearF = (D AtoCenter NEARF);
CenterANearBL = (D AtoCenter NEARBL);
CenterANearBR = (D AtoCenter NEARBR);
CenterAFarL = (D AtoCenter FARL);
CenterAFarR = (D AtoCenter FARR);

CenterAFarM    =    (OR    (P    CenterAFarL)    (P
CenterAFarR));
CenterANearB   =   (OR   (P   CenterANearBL)   (P
CenterANearBR));

CenterAFar = (OR (P CenterAFarL) (P CenterAFarR));
CenterANear    =    (OR    (P    CenterANearL)    (P
CenterANearR));
################################################
```

```

###Probabile puntamento####
Pointing = (D WiiMotePitch POINTING);
################################################

###accelerazione globale###
GAccWeak = (D GAcc WEAK);
GAccStrong = (D GAcc STRONG);
RandomPlus = (D Random PLUS);
RandomMinus = (D Random MINUS);

GAccWeakL = (AND (P GAccWeak) (P RandomPlus));
GAccWeakR = (AND (P GAccWeak) (P RandomMinus));
GAccStrongL = (AND (P GAccStrong) (P RandomPlus));
GAccStrongR    =    (AND    (P    GAccStrong)    (P
RandomMinus));

GAccExist = (D GAcc EXIST);
################################################
```

Predicati multilivello

```
propTanSpeedFF=(D ProposedTanSpeed FAST_FORWARD);
propTanSpeedF = (D ProposedTanSpeed FORWARD);
```

```
propTanSpeedFB=(D ProposedTanSpeed FAST_BACKWARD);
propTanSpeedB = (D ProposedTanSpeed BACKWARD);
```

CANDO

```
GotoGoal=(AND P GoalNotTooClose) (P GoalAligned));
AligntoGoal = (P GoalNotTooClose);

Snaking = (OR (P GAccExist) (P Pointing));
```

```
EscapeIR = (P IRExist);

RobotHit = (P HitV);

StayInArea = (P CenterDFar);
```

WANT

```
GotoGoal = (P GoalNotTooClose);
AligntoGoal = (P GoalNotAligned);

Snaking = (OR (P GAccExist) (P Pointing));
```

```
EscapeIR = (P IRExist);

RobotHit = (P HitV);

StayInArea = (P CenterDFar);
```

Variabili in uscita

TanSpeed	RotSpeed

Insiemi fuzzy delle variabili in uscita

```
(TANSPEED
(SNG (VERY_FAST_FORWARD 30))
(SNG (FAST_FORWARD 10))
(SNG (FORWARD 5))
(SNG (SLOW_FORWARD 3))
(SNG (VERY_SLOW_FORWARD 1))
(SNG (STEADY 0))
(SNG (VERY_SLOW_BACKWARD -1))
(SNG (SLOW_BACKWARD -3))
(SNG (BACKWARD -5))
(SNG (FAST_BACKWARD -10))
(SNG (VERY_FAST_BACKWARD -30))
)
```

```
(ROTSPEED
(SNG (VERY_FAST_RIGHT -40))
(SNG (FAST_RIGHT -20))
(SNG (RIGHT -10))
(SNG (SLOW_RIGHT -5))
(SNG (AHEAD 0))
(SNG (SLOW_LEFT 5))
(SNG (LEFT 10))
(SNG (FAST_LEFT 20))
(SNG (VERY_FAST_LEFT 40))
)
```

Elenco comportamenti

```
( level 1 AligntoGoal AligntoGoal.rul )
( level 1 GotoGoal GotoGoal.rul )

( level 2 Snaking Snaking.rul )
```

```
( level 3 EscapeIR EscapeIR.rul )

( level 4 RobotHit RobotHit.rul )

( level 5 StayInArea StayInArea.rul )
```

GotoGoal

```
(AND (GoalF) (GoalDNear)) => (TanSpeed FORWARD);
(AND(GoalF)(GoalDFar)) => (TanSpeed FAST_FORWARD);
```

```
(AND (GoalB) (GoalDNear)) => (TanSpeed BACKWARD);
(AND(GoalB)(GoalDFar))=> (TanSpeed FAST_BACKWARD);
```

AligntoGoal

```
(GoalANearL) => (RotSpeed LEFT);
(GoalANearR) => (RotSpeed RIGHT);
```

```
(GoalAFarL) => (RotSpeed FAST_LEFT);
(GoalAFarR) => (RotSpeed FAST_RIGHT);
```

Snaking

```
(GAccWeakR) => (RotSpeed RIGHT);
(GAccStrongR) => (RotSpeed FAST_RIGHT);
(GAccWeakL) => (RotSpeed LEFT);
(GAccStrongL) => (RotSpeed FAST_LEFT);

(AND(Pointing)(RandomPlus))=>
                          (RotSpeed FAST_LEFT);
(AND(Pointing)(RandomMinus))=>
                          (RotSpeed FAST_RIGHT);
```

```
(propTanSpeedFF) => (TanSpeed FORWARD);
(propTanSpeedF) => (TanSpeed SLOW_FORWARD);
(propTanSpeedB) => (TanSpeed BACKWARD);
(propTanSpeedFB) => (TanSpeed SLOW_BACKWARD);
```

EscapeIR

```
(IRDirClose)                  =>
            (&DEL.RotSpeedANY)(&DEL.TanSpeed ANY);

(IRDirClose) => (TanSpeed FAST_BACKWARD);
(IRDirNear) => (TanSpeed BACKWARD);
```

```
(IRDirCloseR) => (RotSpeed FAST_RIGHT);
(IRDirCloseL) => (RotSpeed FAST_LEFT);
(IRDirNearR) => (RotSpeed RIGHT);
(IRDirNearL) => (RotSpeed LEFT);
```

RoboHit

```
(HitV) => (&DEL.RotSpeed ANY)(&DEL.TanSpeed ANY);

(AND(NOT(OR(HomeRotV)(HomeDClose)))(HomeAClose))=>
(RotSpeed AHEAD);
(AND(NOT(OR(HomeRotV)(HomeDClose)))(HomeANearL))=>
(RotSpeed LEFT);
(AND(NOT(OR(HomeRotV)(HomeDClose)))(HomeANearL))=>
(RotSpeed LEFT);
(AND(NOT(OR(HomeRotV)(HomeDClose)))(HomeANearR))=>
(RotSpeed RIGHT);
(AND(NOT(OR(HomeRotV)(HomeDClose)))(HomeAFarL))=>
(RotSpeed FAST_LEFT);
(AND(NOT(OR(HomeRotV)(HomeDClose)))(HomeAFarR))=>
(RotSpeed FAST_RIGHT);

(AND(HomeAClose)(HomeDFar))=>
                          (TanSpeedFAST_FORWARD);
(AND(HomeAClose)(HomeDNear))=>
                          (TanSpeed FORWARD);
(AND(NOT(HomeRotV))(AND(HomeANear)(HomeDClose)))=>
(TanSpeed SLOW_FORWARD);
```

```
(AND(HomeRotV) (RobotANearL))=> (RotSpeed LEFT);
(AND(HomeRotV)(RobotANearR))=> (RotSpeed RIGHT);
(AND(HomeRotV)(RobotAFarL))=>
                          (RotSpeed FAST_LEFT);
(AND(HomeRotV)(RobotAFarR))=>
                          (RotSpeed FAST_RIGHT);
(AND(HomeRotV)(RobotAClose))=>
                          (RotSpeed AHEAD);
```

StayInArea

```
(AND(NOT(HitV))(CenterDFar))=>(&DEL.TanSpeed ANY);
(CenterAFarM) => (TanSpeed STEADY);
(CenterANearF) => (TanSpeed FORWARD);
(CenterANearB) => (TanSpeed BACKWARD);

(CenterDVery_Far) =>
           (&DEL.RotSpeed ANY)(&DEL.TanSpeed ANY);
```

```
(AND   (CenterDVery_Far)      (CenterAClose))   =>
(TanSpeed FORWARD);
(AND   (CenterDVery_Far)       (OR   (CenterAFarL)
(CenterANearL))) => (RotSpeed LEFT);
(AND   (CenterDVery_Far)       (OR   (CenterAFarR)
(CenterANearR))) => (RotSpeed RIGHT);
```

BIBLIOGRAFIA

[1] A. Bonarini, M. Matteucci, and M. Restelli.

A novel model to rule behavior interaction

In Proceedings of the 8th Conference on Intelligent Autonomous Systems (IAS-8), page 199 206, Amsterdam, 2004. IOS Press.

[2] Andrea Bonarini, Giovanni Invernizzi, Thomas Halva Labella, and Matteo Matteucci.

An architecture to coordinate fuzzy behaviors to control an autonomous robot.

Fuzzy Sets and Systems, 134, 2003.

[3] Sergio Bittanti

Identificazione dei Modelli e Sistemi Adattativi (sesta edizione), capitolo 5

Pitagora Editrice Bologna, 2004.

[4] Paolo Meriggi.

Processing and Communication Systems for Device Communities.

PhD thesis, Università degli studi di Brescia, 2005.

[5] Simone Ceriani.

Sviluppo di una carrozzina autonoma d'ausilio ai disabili motori

Tesi di Laurea Specialistica, Politecnico di Milano, 2006.

[6] G. Schweighofer, A. Pinz.

Robust Pose Estimation from a Planar Target.

TR-EMT-2005-01, submitted to IEEE Trans. on Pattern Analysis and Machine Intelligence (PAMI), 05/2005.

[7] Kato, H., Billinghurst, M.

Marker Tracking and HMD Calibration for a video-based Augmented Reality Conferencing System.

In Proceedings of the 2nd International Workshop on Augmented Reality (IWAR 99). October, San Francisco, USA, 1999.