Università degli studi di Roma
"La Sapienza"

FACOLTÀ DI INGEGNERIA

Tesi di laurea in Ingegneria Informatica

a.a. 2004-2005
ottobre 2005

Simulazione 3D e applicazioni per robot mobili con UsarSim

Giuliano Polverari

Relatore
prof. Daniele Nardi

Co-Relatori
ing. Alessandro Farinelli
ing. Daniele Calisi

ai miei genitori

Desidero innanzitutto ringraziare il professor Daniele Nardi per avermi dato l'opportunità di sviluppare questa tesi in un campo così stimolante, per la disponibilità mostrata nel seguirne i progressi e per la fiducia instillata in ogni momento del mio lavoro.

Ringrazio inoltre Alessandro Farinelli per avermi fatto da "tutore" lungo tutto lo sviluppo della tesi e per la sua disponibilità a darmi consigli, fornirmi materiale su cui lavorare e seguirmi nelle fasi più critiche.

Ringrazio Daniele Calisi per la sua tranquillizzante reperibilità, per la sua inventiva e per il suo bagaglio inestinguibile di storielle non-*zen*se.

Ringrazio Marco Zaratti per il suo altruismo e per avermi permesso di condividere il comune *patire* su UsarSim.

E' doveroso ringraziare infine tutti i colleghi che in questo periodo, col loro aiuto ed i loro preziosi suggerimenti, hanno accelerato e reso più piacevole lo sviluppo di questo lavoro.

"Hai mai fatto un sogno tanto realistico
da sembrarti vero? E se da un sogno così
non dovessi più svegliarti? Come potresti
distinguere il sogno dalla realtà?"
(Morpheus, dal film Matrix)

Indice

Introduzione

In questa tesi viene presentato UsarSim, un simulatore 3D di ambienti e robot nel contesto del soccorso autonomo. Il simulatore è stato studiato e dotato di alcune funzionalità aggiuntive, come il supporto per la stereovisione, non disponibili inizialmente. E' stata costruita un'interfaccia di basso livello tra UsarSim e la piattaforma di sviluppo robotico Spqr-Rdk. In seguito alla validazione dell'interfaccia, si è utilizzato il sistema complessivo nello sviluppo di applicazioni riguardanti la pianificazione del moto del robot in ambiente sconosciuto e destrutturato. Grazie all'utilizzo del simulatore UsarSim tali applicazioni sono state testate nell'intero arco dello sviluppo in differenti scenari virtuali, ottenendo un migliore impiego del tempo e delle risorse.

Motivazioni

La crescente attenzione negli ultimi anni nei campi dell'intelligenza artificiale e della robotica pone l'accento su temi quali la velocità di sviluppo, l'utilizzo estensivo di apparati di test e le tematiche multiagente.

Tutti questi aspetti portano naturalmente l'attenzione sui simulatori per ambienti tridimensionali ad alto realismo, nella veste di strumenti privilegiati di test in ogni fase del ciclo di sviluppo di sistemi software orientati alla robotica.

Il lavoro svolto si colloca in questo contesto presentando UsarSim: un simulatore 3D potente e versatile, con alle spalle una comunità di svilup-

patori attenta ed eterogenea.

La competizione internazionale *RoboCup* ha ospitato nella scorsa edizione (Osaka 2005) una serie di sessioni dimostrative sull'utilizzo di UsarSim nel contesto della ricerca di vittime in ambienti destrutturati (*RoboCup Rescue*), alle quali ho partecipato quale rappresentante del team SPQR dell'Università "La Sapienza".

La *RoboCup* è una manifestazione nata da un progetto internazionale per promuovere l'intelligenza artificiale, la robotica e gli altri campi a questi correlati. In particolare la *RoboCup Rescue* è una competizione il cui scopo è incoraggiare lo sviluppo e la ricerca in quei campi della tecnologia utilizzati nella ricerca e nel salvataggio di esseri umani in strutture danneggiate da calamità naturali, quali terremoti ed incendi.

A partire dalla prossima edizione (Brema 2006) è prevista l'introduzione nella *RoboCup Rescue Simulation League* della disciplina "3D Simulation", basata sull'utilizzo del simulatore UsarSim.

Descrizione del lavoro svolto

Il lavoro da me svolto è iniziato con uno studio preliminare del simulatore UsarSim e della piattaforma di sviluppo robotico Spqr-Rdk utilizzata nel laboratorio.

In seguito ho creato e testato un'interfaccia di basso livello tra il simulatore e la piattaforma Spqr-Rdk. In questa fase ho apportato alcune modifiche ai sensori virtuali di UsarSim, allo scopo di rendere più realistici i dati forniti, nella forma e nella sostanza. Ho realizzato anche due estensioni del simulatore, inserendo il supporto a nuovi sensori, quali la stereovisione e la Swiss Ranger Camera. Tutti questi miglioramenti apportati ad UsarSim sono stati approvati dalla comunità di sviluppo ed inseriti nella nuova versione del simulatore.

Dopo una fase di validazione dell'interfaccia realizzata ho infine utiliz-

zato il sistema complessivo nella modellazione e realizzazione di applicazioni riguardanti il moto del robot in ambienti destrutturati. Nella prima applicazione ho trattato il problema degli ostacoli invisibili ai sensori, realizzando un sistema di controllo degli stalli del robot, sulla base dello scarto tra la velocità desiderata e quella effettivamente misurata durante il moto. Nella seconda applicazione ho esteso l'algoritmo di pianificazione del moto dell'Spqr-Rdk per tenere conto delle asperità incontrate nel corso dell'esplorazione dell'ambiente.

Risultati conseguiti

L'implementazione del software e i test sul robot hanno mostrato una serie di vantaggi derivati dall'utilizzo della simulazione tridimensionale in ambiente robotico.

Tali vantaggi sono rilevabili sia rispetto all'utilizzo diretto del robot sia rispetto all'utilizzo di un simulatore 2D. Nel primo caso si rilevano un evidente risparmio di tempo, una migliore salvaguardia dello stato del robot e la possibilità di modellare scenari di test complessi e personalizzati. Nel secondo caso si apprezza il grado di fedeltà aggiunto rispetto alla simulazione 2D, dal calcolo dell'odometria all'utilizzo delle telecamere, all'interazione realistica con gli oggetti del mondo simulato.

Le applicazioni che ho sviluppato e testato utilizzando UsarSim sono state impiegate durante la competizione *Real Rescue* per l'annuale appuntamento della *RoboCup*, svoltasi nel 2005 ad Osaka.

Traccia dell'esposizione

Capitolo 1. Viene presentato il contesto USAR e vengono descritti i vantaggi derivati dall'utilizzo di un simulatore 3D in tale contesto.

Capitolo 2. Viene presentato il simulatore UsarSim, con riferimento al-

l'architettura, alle prestazioni ed alle peculiarità, nel contesto dei simulatori 3D orientati alla robotica.

Capitolo 3. Viene descritta la piattaforma software Spqr-Rdk, utilizzata nello sviluppo di applicazioni robotiche nel dipartimento DIS.

Capitolo 4. Viene descritto il lavoro di sviluppo e validazione dell'interfaccia tra UsarSim e Spqr-Rdk, con approfondimento sulle estensioni realizzate in UsarSim per modellare nuovi sensori.

Capitolo 5. Viene presentato il *Sottosistema di Recupero dagli Stalli*, applicazione sviluppata e testata utilizzando UsarSim allo scopo di rilevare gli ostacoli al moto del robot invisibili ai sensori.

Capitolo 6. Viene presentata una seconda applicazione sviluppata utilizzando UsarSim, un'estensione al *Sottosistema di Pianificazione del Moto* di Spqr-Rdk per gestire la presenza di terreno sconnesso.

Capitolo 7. Viene sintetizzato il lavoro svolto e i suoi possibili sviluppi futuri.

Capitolo 1

Simulazione nel contesto USAR

In questo primo capitolo viene presentato il contesto dei robot per il soccorso e vengono analizzati i vantaggi derivati dall'utilizzo della simulazione in tale contesto. Una breve panoramica sui simulatori orientati alla robotica disponibili sul mercato conclude il capitolo.

1.1 USAR: robot per il soccorso

Le operazioni di intervento in ambienti disastrati, rischiosi e in tempi limitati diventano di giorno in giorno più importanti per operazioni antincendio, di salvataggio o di tipo militare. L'utilizzo di robot al posto delle persone nelle attività più pericolose può ridurre drasticamente la perdita di vite umane.

La tecnologia robotica per le missioni di soccorso è una importante sezione della ricerca sulla robotica e sull'intelligenza artificiale. Questo tipo di applicazione coinvolge differenti aree di ricerca, dalla progettazione meccanica all'interpretazione sensoriale e alla percezione: ragionamento automatico, mappatura dell'ambiente, pianificazione di percorsi e individuazione delle vittime.

I robot per il soccorso sono progettati per lavorare a stretto contatto con gli operatori umani, coadiuvandoli durante le missioni. In ogni caso,

l'affidabilità della comunicazione tra robot e operatore umano rappresenta un fattore cruciale nelle missioni di soccorso.

In molte situazioni il robot dovrebbe presentare un certo grado di autonomia per poter agire nell'ambiente in maniera efficace anche in quelle situazioni in cui la comunicazione con gli operatori umani è difficile o impossibile.

Inoltre le capacità autonome come la navigazione sicura, la costruzione della mappa e l'individuazione delle vittime, possono aiutare in maniera consistente gli operatori umani nel controllo del robot.

In questo contesto, le operazioni USAR (*Urban Search and Rescue*) si focalizzano sull'interazione fisica di agenti robotici in ambienti disastrati, in condizioni di disordine disposto casualmente, organizzati in uno standard condiviso.

1.2 Vantaggi derivati dalla simulazione in campo robotico

Lo sviluppo di applicazioni in campo robotico è sottoposto ad una serie di vincoli esterni, dovuti principalmente all'alto valore delle risorse robotiche impiegate. Lo scenario tipico di un laboratorio di ricerca in tale campo è composto di una vasta equipe di ricercatori e tecnici i cui tempi di lavoro sono scanditi dalla disponibilità dei pochi (a volte dell'unico) robot.

Gli apparati robotici hanno un costo ragguardevole, sono estremamente delicati e anche nelle migliori condizioni di utilizzo tendono a deteriorarsi nel tempo: le limitazioni alla loro disponibilità sono di conseguenza inevitabili.

Nel contesto dei robot per il soccorso, con particolare riferimento alle operazioni USAR, sorgono ulteriori problemi per la pericolosità degli ambienti di test: scale, pannelli e macerie possono procurare ai robot danni anche ingenti. Un secondo problema è la difficoltà di creare e modifica-

re ambienti di test condivisi tra tutti gli sviluppatori. Inoltre la varietà e l'imprevedibilità del tipo di ostacoli incontrabili non permette di prevedere adeguatamente le misure di sicurezza per gli apparati utilizzati. L'andamento delle gare internazionali della *RoboCup Rescue* mostra ogni anno quanto anche la migliore pianificazione possa portare ad esiti imprevisti.

La simulazione si inserisce in questo contesto come soluzione ottimale per tutti i problemi sollevati finora. Tramite la simulazione è possibile avere a disposizione in qualsiasi momento un apparato di test completo, personalizzato e riconfigurabile a piacimento.

In simulazione sarà poi reso possibile un set più ampio di test, includendo ambienti di difficile riproduzione nel contesto di un laboratorio e situazioni generalmente trascurate perché possibilmente lesive nei confronti degli apparati robotici reali.

L'utilizzo intensivo del simulatore lascia più ampia disponibilità per i robot reali, che andranno utilizzati solamente per i test finali. La salvaguardia dell'apparato robotico è migliorata anche dal fatto che i test finali avranno per oggetto applicazioni più sicure in quanto già testate in simulazione durante tutto il proprio arco di sviluppo.

Pensando ai contesti multiagente è facile comprendere come un ambiente virtuale consenta l'impiego di gruppi di robot senza limiti di numero (o quasi), facilitando lo studio e lo sviluppo di sistemi di comunicazione e coordinazione in condizioni di testing comode e verosimili.

Come ultimo punto è da sottolineare il vantaggio di poter riutilizzare a fini di debug tutte le informazioni di una simulazione in atto o appena trascorsa. La disponibilità di differenti punti di vista sulla scena simulata e di una visione rallentata delle azioni può rendere più semplice la comprensione degli esiti di ogni test.

In conclusione, la simulazione nel contesto dei robot per il soccorso va incoraggiata senza remore, in quanto apportatrice di vantaggi di ogni sorta,

dai tempi di sviluppo alla salvaguardia dei robot, al testing multiagente, per un miglioramento complessivo delle attività di ricerca robotica.

1.3 Differenti approcci alla simulazione

In questa sezione esamino le possibili distinzioni nel campo dei simulatori software. In mancanza di una sistematizzazione consolidata in questo campo, analizzerò gli ambienti software più in uso nel campo della robotica mobile, cercando di evidenziarne le principali caratteristiche.

Gli attuali simulatori orientati alla robotica possono essere suddivisi in due gruppi in base al tipo di approccio alla simulazione; distingueremo i *simulatori basati su server* ed i *simulatori all-inclusive*.

Simulatori basati su server. Nel primo gruppo ricadono i simulatori in cui un server dedicato simula l'ambiente *in tempo reale* e vari client possono connettersi, generalmente via rete, e testare i propri robot. Secondo questo approccio la simulazione è un flusso inarrestabile ed irripetibile di eventi.
Il dato importante è la netta separazione tra la simulazione del comportamento del robot in un ambiente realistico, demandata al server, e la parte logica che interpreta i dati dei sensori ed impartisce i comandi appropriati, demandata all'operatore o al controller esterni.
Un esempio di questo tipo di simulatori è il software Gazebo, descritto in dettaglio nel paragrafo 1.5.1.

Simulatori all-inclusive. Nei simulatori del secondo gruppo sia l'ambiente che i robot sono gestiti in modo centralizzato, tenendo sotto controllo tutti i parametri disponibili. La simulazione non viene concepita come successione di eventi in tempo reale, quanto piuttosto come un unico evento entro cui navigare in senso temporale, alla velocità e nel verso desiderati.

Questo apprezzabile vantaggio viene scontato con la necessità di introdurre nel simulatore tutti i dati relativi al controllo del robot, precludendo di fatto l'utilizzo di software di controllo esterni o di terze parti.
Come esempio di questa tipologia si può citare il simulatore SimRobots, descritto nel paragrafo 1.5.2.

Nonostante la divisione tra i due gruppi appaia netta, non è raro incontrare simulatori dalle caratteristiche ibride; come esempio di ciò si può citare il simulatore Webots (descritto nel paragrafo 1.5.3), che permette sia una gestione centralizzata della simulazione, sia la possibilità di utilizzare controller esterni tramite TCP/IP.

Indipendentemente dal tipo di approccio alla simulazione, si possono individuare una serie di modalità di utilizzo della simulazione che i vari software possono privilegiare, consentire o meno. Tali modalità possono a mio avviso essere riassunte in:

Modalità realistica Per modalità realistica intendo lo sforzo di riprodurre fedelmente una realtà esistente. Nel contesto USAR, questo approccio alla simulazione è estremamente utile; basti pensare che nella maggior parte dei casi l'ambiente sperimentale di laboratorio è costituito da un solo robot ed una sola arena, con conseguente schedulamento seriale dei test. Un simulatore in grado di fornire la riproduzione digitale dell'ambiente di lavoro permette di condurre i test in maniera indipendente, evitando attese per l'utilizzo del robot e per l'eventuale riconfigurazione dell'arena (test diversi hanno bisogno di ambienti diversi). Un ulteriore vantaggio è dato dal minore carico di lavoro gravante sul robot, con conseguente riduzione dell'usura e dei rischi derivati dai test.

Modalità semplificativa Un secondo approccio alla simulazione è volto a ridurre la complessità dell'ambiente di esecuzione dei test. Questo

approccio risulta utile nella progettazione di strati software ad alta astrazione, ad esempio per gli aspetti di coordinazione multiagente o per la pianificazione topologica del percorso. La modalità semplificativa persegue questo obiettivo presentando ambienti a livello di realismo basso o adattabile alle esigenze.

Modalità proattiva Per modalità proattiva intendo l'utilizzo del simulatore come banco di prova per nuove configurazioni del robot. In questa ottica il simulatore non viene più posto alla fine della catena di sviluppo, bensì all'inizio. Anche l'integrazione di nuovi sensori sul robot può essere studiata e modellata nel dettaglio prima di procedere all'acquisto. Il simulatore diventa un ausilio nella pianificazione delle spese e dell'utilizzo complessivo delle risorse hardware.

1.4 L'*assunzione 2D* e la terza dimensione

Storicamente il panorama dei simulatori orientati alla robotica è legato agli ambienti a due dimensioni. Anche in tempi recenti, nonostante l'avvento di numerosi simulatori per ambienti tridimensionali, la preferenza dei ricercatori rimane per i simulatori 2D.

Questo attaccamento, a mio avviso, deriva da un insieme di fattori, tra i quali la ritrosia verso le nuove tecnologie (la cosiddetta inerzia tecnologica) è certamente il minore.

Il fattore fondamentale è senza dubbio l'utilizzo generalizzato dell'assunzione teorica di mondo planare, la cosiddetta *assunzione 2D*, di cui cito la definizione data da Leonard e Durrant-Whyte in [6]

> Assumiamo che la geometria dell'ambiente tridimensionale visitato sia ortogonale al piano orizzontale in cui il robot si muove.

Questo vincolo, che semplifica notevolmente i task che il robot dovrà affrontare, è molto stringente. Innanzitutto si assume che un piano su cui il robot

cammina esista (si pensi ad un robot che volesse camminare su delle assi di legno sconnesse), che tale piano sia orizzontale (potrebbero esserci rampe, salite, discese, scale), infine che gli oggetti incontrati siano "proiettabili su una mappa vista dall'alto" (un tavolo o una sedia invece sono oggetti la cui geometria non rispetta l'assunzione).

Assumendo questo vincolo i task di localizzazione, navigazione e mapping sono stati per anni trattati esclusivamente con sensori tipo Sonar o Laser, tipicamente sistemati in posizioni fisse davanti, dietro e ai lati del robot, con orientazione parallela al piano di navigazione.

Volendo rimuovere qualcuna delle ipotesi contenute nell'*assunzione 2D*, laser e sonar in posizione fissa si trovano a portare un contenuto informativo insufficiente per rilevare la forma dell'ambiente circostante. Si supponga ad esempio di avere di fronte a sè un oggetto di forma irregolare, come una sedia: un anello di sonar posto ad un'altezza di 20-30 cm probabilmente non riuscirà neanche a rilevarne la presenza, mentre un laser probabilmente crederà di avere di fronte a sè quattro pali in corrispondenza delle gambe della sedia. Si pensi ora di esaminare un ambiente non pianeggiante: la presenza di una salita di fronte al robot sarà interpretata probabilmente come un muro, mentre durante l'attraversamento di una discesa il pavimento stesso viene rilevato come una ostruzione. Se poi il piano dove il robot si muove non esiste, stiamo parlando di una situazione di estrema difficoltà, in cui non è necessario conoscere solo che cosa si trova nelle estreme vicinanze del robot, ma anche ciò che si trova al di sotto di lui o al di sopra.

La trattazione precedente spiega la ritrosia dell'ambiente di ricerca ad abbandonare l'*assunzione 2D*. La conseguenza di questo atteggiamento è evidente: per testare e sperimentare task di localizzazione, navigazione e mapping sotto l'*assunzione 2D* appare sufficiente, se non più comodo, utilizzare simulatori di ambienti bidimensionali, che presentano tra l'altro il vantaggio di occupare poche risorse di calcolo.

Eppure, a mio avviso, la simulazione 2D è una tecnica destinata a sparire entro breve, per una serie di motivazioni:

Realismo della simulazione. Il livello di realismo offerto dai simulatori 3D non è in alcun modo paragonabile a quello dei loro predecessori. Un modello tridimensionale dell'ambiente permette la simulazione della gravità, del moto (a livello di motori, attuatori e ruote) e di complesse interazioni dinamiche tra gli oggetti.

Resa grafica. Dal punto di vista meramente grafico, l'ausilio di punti di vista sull'ambiente posizionabili e ruotabili a piacimento permette una migliore comprensione della scena simulata e degli esiti dei test.

Quantità di sensori simulabili. I simulatori 3D offrono la possibilità di includere un numero maggiore di sensori, tra cui telecamere e sensori inerziali.

Fusione sensoriale. La disponibilità di una più ampia gamma di sensori sposta l'attenzione sulle tecniche di fusione sensoriale, la cui simulazione era in precedenza limitata ai soli sensori 2D (laser, sonar), nell'ulteriore assunzione di orientazione sullo stesso piano.

Ad ulteriore conferma della superiorità della simulazione 3D rispetto alle tecniche precedenti, è interessante notare come i più importanti simulatori 2D in ambiente robotico prevedano, nelle versioni più recenti, il supporto di estensioni di tipo tridimensionale. Ad esempio la piattaforma di simulazione 2D Player/Stage è stata dotata recentemente del modulo Gazebo (descritto nel paragrafo 1.5.1), per la simulazione di ambienti in tre dimensioni.

In conclusione, l'utilizzo dei simulatori 3D in campo robotico è a mio avviso destinato a soppiantare in breve tempo le tecniche legate ad ambienti bidimensionali. Nel prosieguo della trattazione saranno considerati solamente simulatori 3D.

1.5 Panoramica dei simulatori 3D per la robotica

Un simulatore 3D è composto essenzialmente da un motore grafico e da un motore fisico. Questi due strumenti affrontano problemi differenti, quello della resa grafica e osservabilità dell'ambiente e quello della credibilità fisica degli eventi riprodotti.

Nell'ottica di perseguire questi diversi fini, i due motori utilizzano nozioni e tecniche provenienti da campi diversi e disgiunti e prediligono differenti risorse di calcolo.

Dal lato della simulazione grafica, esistono eccellenti driver e strumenti provenienti dal campo di ricerca della realtà virtuale. Dal lato della simulazione fisica non mancano buoni motori basati sulle dinamiche dei corpi rigidi, tra cui ad esempio ODE[1], un motore dotato di un'interfaccia grafica basata su finestre.

Purtroppo però risulta difficile trovare prodotti che riescano ad integrare con equilibrio un buon motore grafico ed un buon motore fisico. La maggior parte dei prodotti si appoggia ad ODE per la simulazione delle dinamiche fisiche e propone soluzioni ad hoc per la componente grafica.

L'attuale panorama dei simulatori 3D per la robotica risulta quindi attualmente esiguo e frastagliato, composto tra prodotti con finalità e approcci differenti e per questo difficilmente confrontabili.

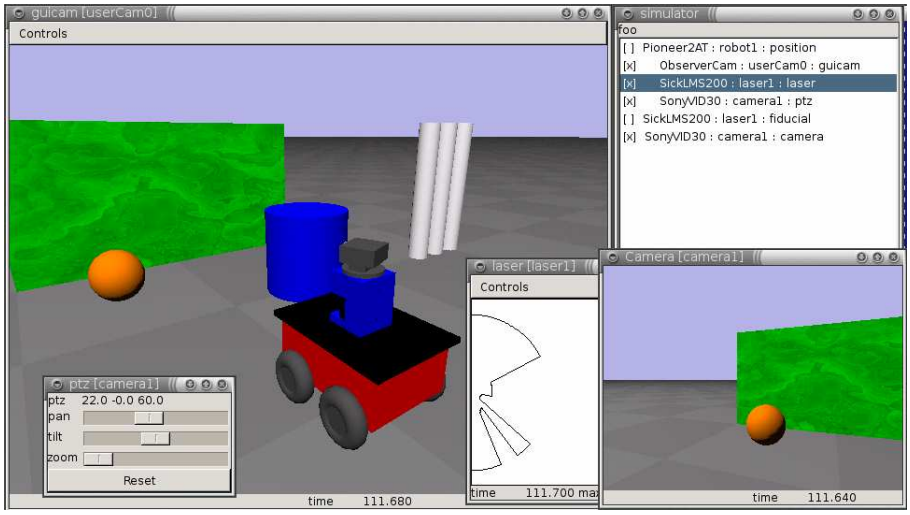
Nel seguito presento gli aspetti peculiari dei simulatori più importanti. La descrizione di UsarSim, il simulatore oggetto del mio lavoro, viene rimandata al capitolo 2.

1.5.1 Gazebo

Gazebo[2] è il simulatore 3D inserito nel progetto opensource Player/Stage. Nell'ambito del progetto i programmi principali sono Player, che simula

[1] ODE, Open Dynamics Engine `www.ode.org`
[2] Gazebo `http://playerstage.sourceforge.net`

Figura 1.1: Simulazione 3D con Gazebo



un robot ed i suoi sensori, e Stage, che permette la simulazione di grandi gruppi di robot in un ambiente bidimensionale con basso grado di realismo.

In questo contesto Gazebo si inserisce come l'estensione di Stage per ambienti tridimensionali, con l'obiettivo di simulare un piccolo gruppo di robot con un alto grado di realismo. Nella Figura 1.1 si può osservare la resa visiva del motore grafico.

Il simulatore utilizza l'approccio *basato su server* (come definito in 1.3). La comunicazione col server Gazebo può essere stabilita utilizzando Player come tramite oppure direttamente, attraverso libreria `libgazebo` fornita col simulatore.

Vengono forniti diversi tipi di sensori e robot già modellati. Come nota di rilievo si può citare l'inserimento nell'ultima release del modello di una stereo camera. Gazebo è rilasciato come software libero, sotto licenza GPL (GNU Public License).

1.5.2 SimRobot

SimRobot[3] è il simulatore 3D sviluppato dall'Università di Brema e utilizzato correntemente dal GermanTeam[4].

L'applicazione è di tipo *all-inclusive*, per sfruttare il vantaggio di poter interrompere, riprendere e modificare la simulazione in qualsiasi momento; ad ogni passo del processo di elaborazione viene aggiornato l'ambiente ed eseguito un ciclo di sensing-decisione-azione nel controllore del robot, che è completamente integrato. Molto interessante è la possibilità di manipolazione diretta degli attuatori ed interazione col mondo simulato.

Come la maggior parte dei simulatori, anche SimRobot utilizza il motore fisico ODE, mentre il sistema di visualizzazione è basato su OpenGL. Per la specifica dei robot e dell'ambiente viene utilizzato il linguaggio RoSiML[5], sviluppato dalla stessa Università di Brema in collaborazione col Fraunhofer Institute for Autonomous Intelligent Systems.

L'interfaccia utente è molto curata e permette a tempo di esecuzione di navigare l'ambiente utilizzando varie viste (sottoinsiemi della scena) a differenti gradi di dettaglio, di visualizzare il modello fisico degli oggetti (basato su primitive ODE), di aggiungere, spostare e ruotare interattivamente tutti gli oggetti visualizzati.

Per quanto riguarda i sensori, viene fornito un insieme di classi generiche comprendente una telecamera, un sensore di distanza, un sensore di collisione ed un rilevatore dello stato di un attuatore; da queste classi l'utente può derivare i sensori che intende utilizzare.

Il codice di SimRobot viene rilasciato come software open source. Ulteriori informazioni sul simulatore possono essere reperite in [8].

[3] SimRobot `http://www.informatik.uni-bremen.de/simrobot/index_e.html`

[4] German Team `http://www.germanteam.org`

[5] RoSiML - Robot Simulation Markup Language `http://www.tzi.de/spprobocup/RoSiML.html`

1.5.3 Webots

Webots[6] è un simulatore 3D prodotto e commercializzato dalla Cyberbotics Ltd; una buona introduzione al simulatore è disponibile in [7]. La release 5, attualmente in commercio, presenta un ambiente integrato per la modellazione, la programmazione e la simulazione di robot mobili. Come motore fisico viene utilizzato ODE.

Il modello di sviluppo preferenziale proposto in Webots prevede la programmazione dei sistemi di controllo direttamente all'interno del simulatore; in questo modo Webots riesce a coprire tutti gli aspetti dello sviluppo di un'applicazione robotica. Le librerie fornite permettono di trasferire sui robot reali i programmi di controllo sviluppati in Webots, nel momento in cui siano ritenuti stabili ed affidabili, tramite un processo di cross-compilazione.

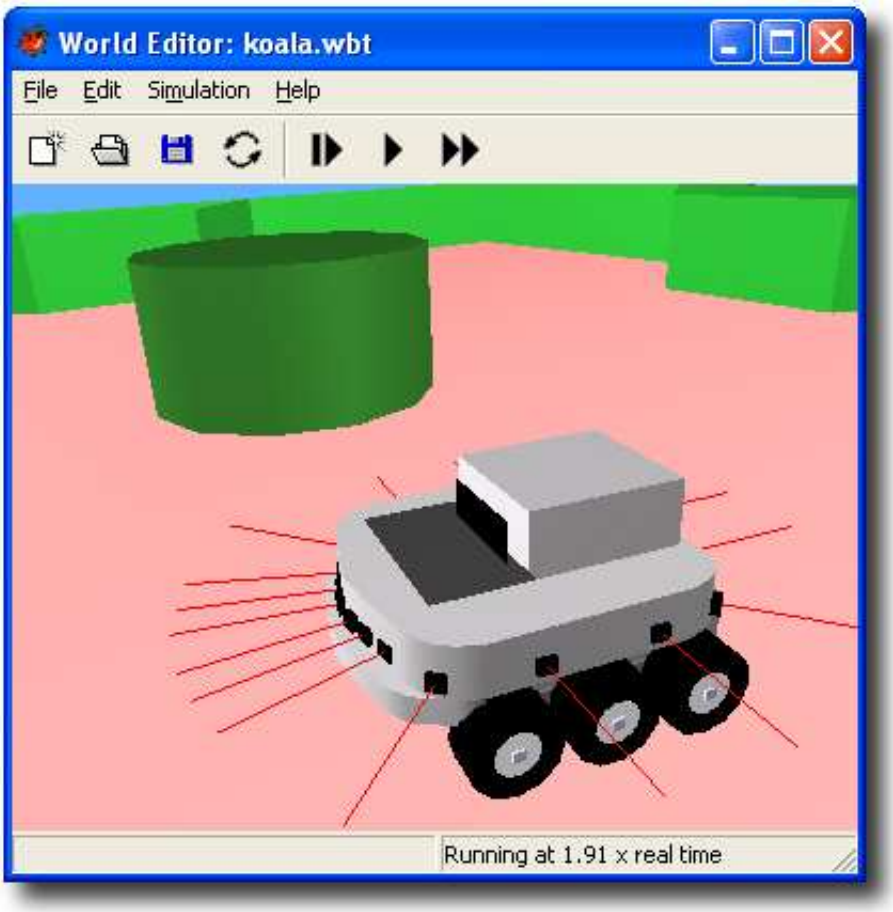
In virtù della gestione centralizzata della simulazione, Webots può essere inserito di diritto nella categoria *all-inclusive*, nonostante permetta l'utilizzo di controller esterni attraverso connessioni basate su TCP/IP.

Sono forniti diversi modelli di robot quali ad esempio Aibo, Khepera ed il Pioneer 2. Il programma ha alle spalle uno sviluppo di sette anni (è tra i più longevi tuttora in commercio) ed è disponibile per le piattaforme software Windows, Linux e Mac OS X. Tra le feature più interessanti è possibile citare:

- la possibilità di rallentare la velocità della simulazione fino alla modalità passo-passo, per visionare al massimo dettaglio il comportamento dei robot;
- la possibilità di muovere e ruotare a tempo di esecuzione gli oggetti presenti nella simulazione, allo scopo di facilitare i test interattivi;
- la possibilità di inserire all'interno della simulazione un programma con funzione di supervisore, al quale possono essere associate azioni quali ad esempio la movimentazione degli oggetti, la spedizione di

[6] Webots `http://www.cyberbotics.com`

Figura 1.2: Simulazione 3D con Webots



messaggi ai robot, la memorizzazione delle traiettorie e l'aggiunta di nuovi oggetti e robot.

Nella Figura 1.2 viene mostrata l'interfaccia verso il mondo simulato. Si possono notare i due pulsanti rallentare o accelerare la velocità della simulazione.

1.5.4 UchilSim

UchilSim[7] è un simulatore robotico sviluppato dall'Università del Cile. E' progettato specificamente per la RoboCup Four-Legged League. Il simulatore si appoggia a ODE per la simulazione delle dinamiche.

L'ambiente ed i robot sono descritti in una struttura VRML con estensioni per gli elementi della simulazione e gli attributi fisici.

1.5.5 ÜberSim

Il simulatore ÜberSim[8] è focalizzato principalmente sui robot dotati di visione. Ha una architettura *basata su server* poggiata sul protocollo TCP/IP ed utilizza ODE come motore fisico.

La release corrente presenta solamente due sensori predefiniti, una telecamera ed un inclinometro. Attualmente non è fornito di alcuna interfaccia utente di tipo grafico per interagire con i robot e l'ambiente a tempo di simulazione.

[7] UchilSim `http://www.robocup.cl/uchilsim/`
[8] ÜberSim `http://www.cs.cmu.edu/~robosoccer/ubersim/`

Capitolo 2

Presentazione del simulatore UsarSim

In questo capitolo viene presentato UsarSim, un simulatore 3D ad alta fedeltà orientato al soccorso robotico in ambienti disastrati. Viene presentata una breve panoramica del sistema, della sua storia e della comunità che lo sviluppa ed utilizza. Successivamente vengono descritti l'architettura del simulatore e l'ambiente di simulazione, con riferimento alle Arene, ai Robot ed ai Sensori modellati. Un approfondimento sui vantaggi e limiti e sulle prestazioni di UsarSim conclude il capitolo.

2.1 Panoramica del sistema

UsarSim, presentato in [2, 3], è un simulatore 3D orientato al soccorso robotico, sviluppato come strumento di ricerca all'interno di un progetto della National Science Foundation[1] (NSF), riguardante lo studio di Robot, Agen-

[1]La National Science Foundation (NSF) è un'agenzia federale indipendente creata dal Congresso americano nel 1950 "per promuovere il progresso della scienza; per migliorare il benessere e la prosperità della nazione; per assicurare la difesa nazionale". Con un budget annuale di circa 5.5 miliardi di dollari, la NFS fornisce i fondi per circa il 20 per cento di tutte le ricerche di base federalmente supportate, condotte dalle università e dai college d'America. La NSF è raggiungibile online al sito www.nsf.org.

ti e Team di operatori nel contesto di Urban Search And Rescue (USAR, operazioni di Ricerca e Soccorso in ambiente Urbano).

In UsarSim viene riprodotto fedelmente un ambiente di soccorso robotico, includendo una serie di arene, di robot e tutta la sensoristica necessaria. Il professor Michael Lewis, dell'Università di Pittsburgh, costituisce la guida ed il principale punto di riferimento del team di sviluppo di UsarSim; la manutenzione del codice è affidata allo sviluppatore principale Jijun Wang, che cura il rilascio delle varie release del simulatore. La release corrente può essere scaricata liberamente dal sito `http://usl.sis.pitt.edu/ulab/usarsim_download_page.htm`.

Il simulatore è stato progettato come un'estensione del motore di gioco *Unreal Engine*, un software commerciale multipiattaforma orientato al gaming FPS (First Person Shooting) multigiocatore, sviluppato dalla Epic Games[2].

UsarSim demanda completamente all'*Unreal Engine* il rendering grafico della scena tridimensionale e la simulazione delle interazioni fisiche tra gli oggetti. In questo modo, lasciando gli aspetti più difficili della simulazione ad una piattaforma commerciale in grado di offrire un rendering visuale superiore ed una buona modellazione fisica, tutti gli sforzi nello sviluppo di UsarSim sono concentrati sui compiti specifici della robotica, quali la modellazione di ambienti, sensori, sistemi di controllo e strumenti d'interfaccia.

Inoltre lo sviluppo è facilitato dall'utilizzo dei software di editing avanzato integrati nel motore di gioco, quali l'editor grafico *Unreal Editor* ed il linguaggio di scripting interno *Unreal Script*.

2.1.1 RoboCup Rescue

La *RoboCup Rescue* è una competizione internazionale annuale il cui scopo è quello di incoraggiare lo sviluppo e la ricerca in quei campi della tecno-

[2]Epic Games `http://www.epicgames.com`

logia utilizzati nella ricerca e nel salvataggio di esseri umani in strutture che hanno subìto gli effetti di un terremoto, edifici in fiamme, incidenti sotterranei e così via.

Attualmente, durante le competizioni, vengono preparate delle "arene" che riproducono l'interno di edifici colpiti da terremoti o incendi, secondo gli standard *NIST Usar Test Facility* descritti in [1]. All'interno delle arene sono posizionati manichini, che costituiscono le vittime da individuare. Ogni squadra in gara presenta uno o più robot che dovranno esplorare l'ambiente in modalità autonoma o teleoperata.

Lo sviluppo, in questo ambito, di agenti in grado di agire in maniera autonoma ed esplorare tali ambienti, individuando le eventuali vittime, è una sfida interessante, in quanto coinvolge aspetti quali la mappatura dell'ambiente, la localizzazione del robot, l'individuazione degli ostacoli e delle vittime e la necessità di predisporre una modalità di esplorazione completamente autonoma in caso di impossibilità di comunicare col robot.

A partire dalla prossima edizione (Brema 2006) è prevista l'introduzione nella *Rescue Simulation League* della disciplina "3D Simulation", basata sull'utilizzo del simulatore UsarSim.

2.1.2 Numerazione convenzionale delle versioni

Nello stadio attuale di sviluppo di UsarSim non è prevista una numerazione sistematica delle versioni. Per sopperire a questa mancanza ho scelto una numerazione convenzionale per le versioni del simulatore citate nella mia trattazione.

La convenzione è la seguente:

- **versione 0.1:** identifica la versione di UsarSim che ho utilizzato dall'inizio del mio lavoro;
- **versione 0.2:** identifica la versione di UsarSim utilizzata nell'esibizione tenuta all'interno della manifestazione RoboCup Rescue 2005,

ad Osaka; include la mia modifica al laser scanner, presentata nel paragrafo 4.4.1;

- **versione 0.3:** identifica la versione corrente del simulatore al momento della pubblicazione della tesi, ratificata dal team di sviluppo di UsarSim; include il mio lavoro sulla Stereovisione, presentato nel paragrafo 4.3.1.

2.2 Architettura di UsarSim

Dal punto di vista architetturale UsarSim è strutturato come sistema client-server. L'architettura del simulatore è mostrata nella Figura 2.1, nella quale sono isolate la parte client (in alto) e la parte server (in basso), connesse da uno strato di rete.

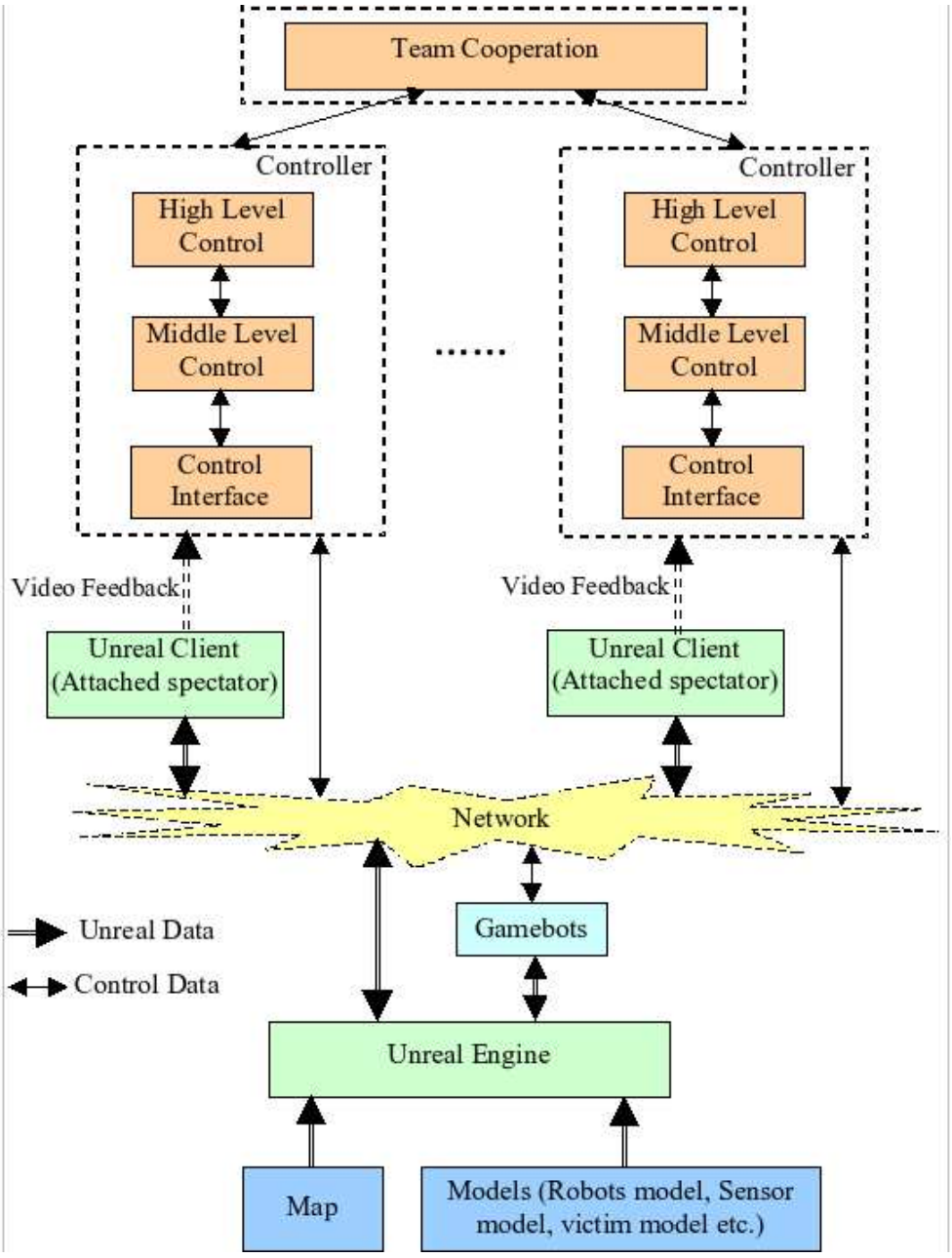
Lato client. Della parte client fanno parte il *Controller* dell'utente, racchiuso nei rettangoli tratteggiati, e l'*Unreal Client*, che fornisce il feedback video relativo ai punti di vista disponibili (solitamente, una camera su ogni robot robot più una visuale esterna).

Strato di rete. Tutti i client scambiano dati col server attraverso lo strato di rete, basato sul protocollo TCP/IP.

Lato server. Dalla parte server la gestione della simulazione è basata sull'*Unreal Engine*, il motore di gioco vero e proprio, che si avvale della *mappa* dell'arena corrente, contentente tutti i dati dell'ambiente di simulazione, e dei *modelli* utilizzati nella simulazione, tra cui i robot, i sensori e le vittime.

La comunicazione con il lato client è bipartita: i dati relativi al feedback grafico vengono inviati direttamente in rete agli *Unreal Client* connessi; il dialogo con i *Controller* è invece gestito tramite il protocollo di comunicazione *Gamebots*.

Figura 2.1: Architettura di UsarSim



Nel seguito verrà approfondita la descrizione del lato server, con particolare riferimento all'*Unreal Engine*, al protocollo *Gamebots* ed ai *Controller* utilizzabili.

2.2.1 Unreal Engine

Il motore di gioco alla base di UsarSim, l'*Unreal Engine*[3], è rilasciato dalla Epic Games all'interno del gioco Unreal Tournament (le versioni di riferimento sono UT2003[4] ed UT2004[5]), disponibile per i sistemi operativi MS Windows, Linux e Mac OS X. In alternativa è utilizzabile il software Unreal Engine Runtime (versione di riferimento UE2R[6]), allo stato attuale disponibile esclusivamente per il sistema operativo MS Windows.

UsarSim è eseguibile sulle tre piattaforme UT2003, UT2004, UER2, anche se il supporto ad UE2R è limitato.

L'*Unreal Engine* è un sistema completo di sviluppo e simulazione di ambienti tridimensionali, composto da:

- un renderer grafico 3D, *Unreal Client*, in grado di fornire visuali egocentriche (poste sul robot) ed exocentriche (in terza persona) della simulazione, che può essere utilizzato per esigenze di debug e di sviluppo;
- un motore per le interazioni fisiche, *Karma Engine*, che simula la gravità e le interazioni tra gli oggetti;
- un tool di authoring 3D, *Unreal Editor*, per la modellazione di mappe ed attori;
- un linguaggio di scripting, *Unreal Script*, per modificare agevolmente il comportamento del sistema.

[3] Unreal Engine `http://unreal.epicgames.com`
[4] Unreal Tournament 2003 `http://www.unrealtournament2003.com`
[5] Unreal Tournament 2004 `http://www.unrealtournament2004.com`
[6] Unreal Engine 2 Runtime `http://edn.epicgames.com/Two/UnrealEngine2Runtime22262001`

L'*Unreal Engine* è considerato lo standard di fatto tra i game engine adattati alla ricerca scientifica. Nel parco di progetti che ne fanno uso si possono citare ad esempio:

- display panoramici immersivi, con tecniche multi-schermo (progetto CaveUT), a cura di J. Jacobson, *Università di Pittsburgh*;
- ambienti di test per Intelligenza Artificiale, a cura di John Laird, *Università del Michigan*;
- architetture cognitive (progetto Act-R), a cura di J. Anderson e C. Lebiere, *Carneige Mellon University*;
- generazione di personaggi sintetici, a cura di K. Sycara, J. Hodgins e G. Sukanthar, *Carneige Mellon University*;
- modellazione architetturale (progetto Virtual Reality Notre Dame), a cura di V.J. De Leon, *Digitalo Studios*;
- modellazione militare a scopo tattico (progetto MOVES per la US Army), a cura di M.Zyda, *Naval Postgraduate School*;
- framework per il supporto di simulazioni interattive distribuite in ambiente militare (progetto UTSAF), a cura di J. Manojlovich, *Università di Pittsburgh.*

2.2.2 Gamebots, il protocollo di comunicazione

Il protocollo di comunicazione utilizzato dall'*Unreal Engine* è proprietario; questo rende difficile l'accesso ad *Unreal Tournament* da parte di applicazioni esterne.

Gamebots[7] è un software sviluppato dai ricercatori dell'ISI[8] della *University of Southern California* allo scopo di consentire la comunicazione con

[7]Gamebots `http://www.planetunreal.com/gamebots`
[8]ISI, Information Sciences Institute `http://www.isi.edu`

l'*Unreal Engine*, in un contesto di rete, tramite l'apertura di una connessione (*socket*) di tipo TCP/IP, utilizzabile da applicazioni di terze parti. Per default il *socket* viene aperto sulla porta 3000.

UsarSim utilizza *Gamebots* come protocollo di comunicazione tra l'*Unreal Engine* ed i *Controller* esterni, e vi applica alcune modifiche per supportare le proprie classi di dati.

La comunicazione è basata sullo scambio di dati di testo semplice, che seguono il formato

TIPO {Segmento1} {Segmento2} ...

in cui

TIPO: è una sigla, scritta in maiuscolo, che identifica la tipologia dei dati che seguono; nella Tabella 2.1 sono elencati i tipi di dati supportati in UsarSim. Esempi: INIT, STA, SEN, DRIVE;

Segmento: è una lista di coppie nome/valore, in cui nome e valore sono separati da uno spazio; il valore può contenere varie informazioni, separate da punteggiatura arbitraria (in genere da virgole) purché non siano inclusi spazi. Esempi: {Name *Left* Range *144.19*}, {Orientation *0,0,200*}, {LightIntensity *100*}, {Battery *80.0*}.

I dati scambiati in UsarSim vengono suddivisi in base alla sorgente, tra i *messaggi,* inviati dal robot e dai sensori, ed i *comandi*, inviati dal *Controller* remoto. Attualmente esistono vari tipi di messaggi e comandi, elencati nella Tabella 2.1. Tutti i dati (messaggi e comandi) inviati sono terminati dalla stringa di chiusura `[CR][LF]`[9].

2.2.3 Controller

UsarSim si presta ad essere utilizzato per testare sistemi di controllo e coordinamento robotici; di conseguenza lo sviluppo dell'interfaccia col si-

[9] I caratteri ASCII `[CR][LF]` (*carriage return* e *line feed*) rappresentano uno standard utilizzato per determinare la terminazione di una riga di testo.

Tabella 2.1: Tipi di dati (messaggi e comandi) utilizzati in UsarSim

Messaggi	**Utilizzo**
STA	informazioni sullo stato del robot
SEN	dati forniti dalle letture dei sensori
GEO	informazioni sulla geometria dei sensori
CONF	informazioni sulla configurazione dei sensori
Comandi	**Utilizzo**
INIT	inserisce un robot nell'arena, in una data posizione
DRIVE	controlla il movimento del robot; il controllo può agire sull'insieme delle ruote o su un singolo giunto (con ordine *zero*, *uno* o *due*, come descritto nel par. 2.3.2)
CAMERA	controlla i movimenti della telecamera, tramite il sistema pan-tilt (con vario ordine, come per il comando DRIVE)
SET	viene utilizzato per attivare specifici comportamenti dei sensori, ad esempio l'attivazione del laser scanner o il reset della posizione della telecamera
GETGEO	richiede un'informazione sulla geometria di un sensore
GETCONF	richiede un'informazione sulla configurazione di un sensore

mulatore è demandato all'utente. In questo contesto, vengono denominate *Controller* tutte le applicazioni esterne che utilizzano il simulatore.

Le operazioni usuali all'interno di un *Controller* sono le seguenti:

- inizialmente il *Controller* apre un socket tramite il quale si connette all'*Unreal Engine*;
- poi il *Controller* invia il comando per inserire un robot in UsarSim;
- a questo punto il *Controller* entra in un ciclo infinito in cui ad ogni passo riceve i dati dai sensori ed invia comandi per controllare il robot.

L'architettura client/server di Unreal permette di aggiungere molteplici robot nella simulazione. Comunque, considerato che ogni robot utilizza un proprio socket per comunicare, il *Controller* deve creare una diversa connessione per ognuno dei robot impiegati.

Tra gli esempi di implementazione di un *Controller*, UsarSim mette a disposizione un'interfaccia di test denominata SimpleUI, utilizzabile per controllare il robot manualmente.

Inoltre in UsarSim è inserito un *plugin* per connettersi a Pyro[10], un software scritto in Python comprendente librerie, GUI e driver a basso livello per applicazioni robotiche. Tramite il *plugin* è possibile utilizzare Pyro per controllare il robot nella simulazione.

UsarSim fornisce anche un pacchetto di driver per Player[11], un robot device server che permette all'utente un controllo semplice e completo sui sensori e gli attuatori del robot. Tramite i driver forniti con UsarSim, è possibile controllare il robot nella simulazione attraverso Player come se fosse un device reale.

In conclusione, l'interfaccia può essere diretta al protocollo *Gamebots* oppure al middleware Pyro o Player. Sebbene la connessione diretta tramite *Gamebots* risulti generalmente la scelta più azzeccata in termini di velocità e semplicità d'esecuzione, l'utilizzo di Pyro o Player come *bridge*

[10]Pyro `http://pyrorobotics.org`
[11]Player `http://playerstage.sourceforge.net`

(programma ponte) verso UsarSim permette un maggiore grado di disaccoppiamento (ad esempio non c'è necessità di realizzare conversioni con le *Unreal Units*) e l'utilizzo di strumenti di visualizzazione e controllo altrimenti non disponibili.

2.3 Panoramica dell'ambiente di simulazione

Nella terminologia di *Unreal* tutti gli oggetti che, come un robot, un sensore o una vittima, possiedono comportamenti individuali, sono definiti Attori; i luoghi in cui gli Attori si muovono ed interagiscono sono chiamati Arene.

2.3.1 Le Arene

La ricostruzione dell'ambiente gioca un ruolo molto importante nella simulazione: l'Arena è il necessario contesto senza il quale la simulazione non avrebbe senso. In UsarSim vengono fornite tre Arene (Gialla, Arancione e Rossa), che riproducono ambienti disastrati a difficoltà progressiva, secondo gli standard NIST Usar Test Facility descritti in [1].

Arena Gialla (Figura 2.2): è la più semplice da attraversare, dotata di un ampio piano pavimentato con muri perpendicolari ed ostacoli di difficoltà moderata; priva di particolari requisiti di agilità, offre un ambiente strettamente planare (2D), con ostacoli e obiettivi facilmente raggiungibili tramite i sensori più semplici (laser, sonar, camera);

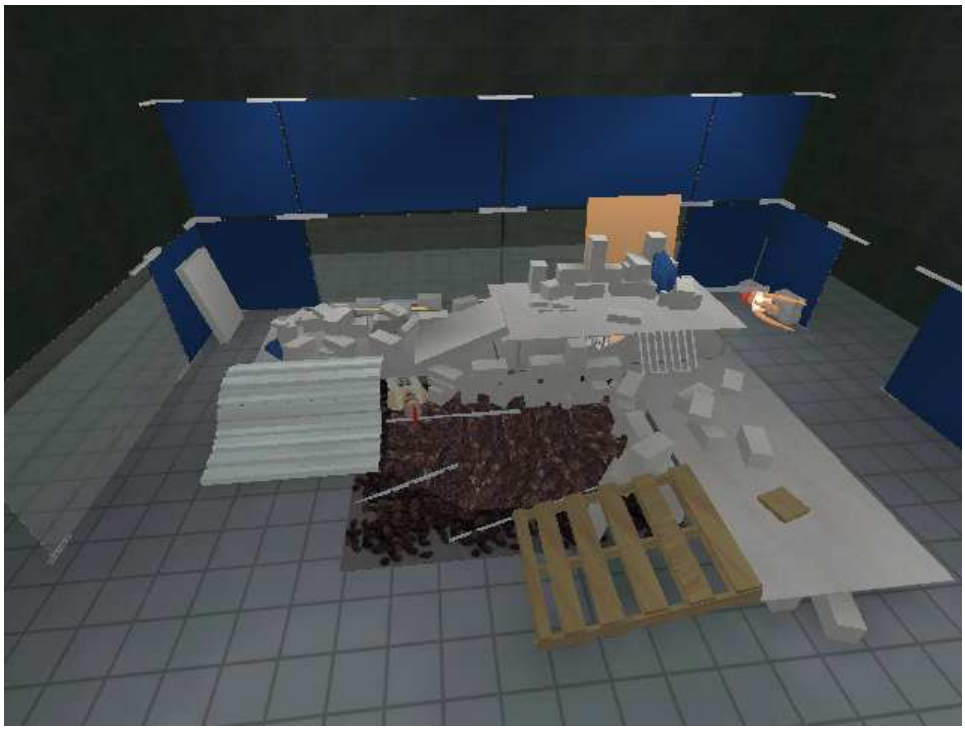
Arena Arancione (Figura 2.3): è un'arena a due piani con ostacoli impegnativi quali una scala a gradini, una rampa ed un pozzo; presenta un ambiente ad alta complessità spaziale (3D) la cui pavimentazione è resa irregolare da carta, cenere e detriti;

Arena Rossa (Figura 2.4): è la più difficile da attraversare in quanto, a fronte di una minore complessità spaziale (manca il secondo piano

Figura 2.2: Arena Gialla

Figura 2.3: Arena Arancione

Figura 2.4: Arena Rossa



utilizzato nell'Arena Arancione), fornisce un ambiente completamete destrutturato; sono presenti cumuli di oggetti problematici quali blocchi di cemento, lastre d'acciaio, buste di plastica e tubi.

Nella versione attuale di UsarSim viene inoltre fornita l'arena parzialmente texturizzata dell'ambiente di riferimento *Nike Silo*.

Le mappe vengono modellate all'interno del tool Unreal Editor, in grado di importare modelli dai più diffusi programmi di authoring 3D (Autocad, 3D Studio, ProEngineer). Per modellare macerie ed ostacoli possono essere utilizzati materiali diversi, tra cui vetri, specchi, nastri arancioni di sicurezza.

Tra gli oggetti predefiniti modellati in UsarSim il più importante e complesso è il modello della vittima[12], in grado di compiere alcune azioni, come muovere le braccia e le gambe o emettere suoni in richiesta d'aiuto.

Riguardo i movimenti, possono essere definiti fino ad 8 segmenti del corpo che implementano mosse indipendenti per velocità e raggio d'azione. Un esempio di configurazione della vittima è mostrato in Figura 2.5.

2.3.2 I Robot

I robot sono gli Attori più importanti delle scene modellate in UsarSim. Il simulatore mette a disposizione sei modelli di robot su ruote già implementati: i modelli *P3-AT* e *P2-DX* della *ActivMedia Robotics*[13], l'*ATVR-Jr* della iRobot, il *Personal Exploration Rover* ed il *Corky* progettati dall'*Usar Team* della *Carneige Mellon University*[14], ed una generica automobile a quattro ruote.

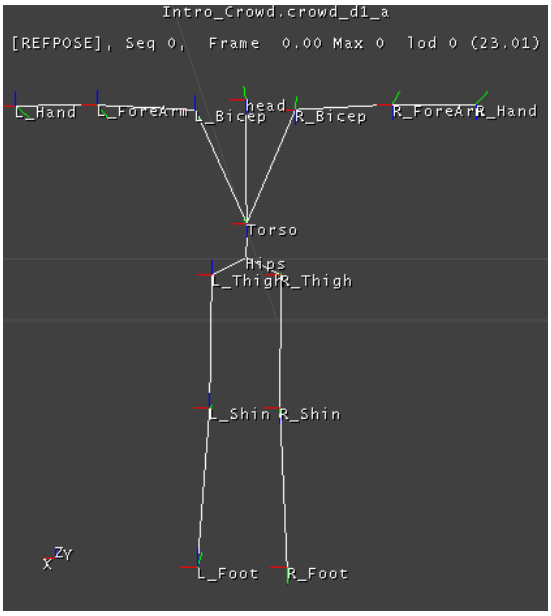
Accanto a tali modelli è possibile implementarne di nuovi, non necessariamente su ruote; a breve, ad esempio, sarà disponibile il modello del

[12] Il modello della vittima è importabile nell'*Unreal Editor* dalle *Actor Classes*, tramite il percorso `Actor\Pawn\UnrealPawn\xIntroPawn\USARVictim`.

[13] ActivMedia Robotics `http://www.activrobots.com`

[14] Carneige Mellon University `http://www.cmu.edu`

Figura 2.5: Modello della vittima in UsarSim



robot su zampe *Sony Aibo*.

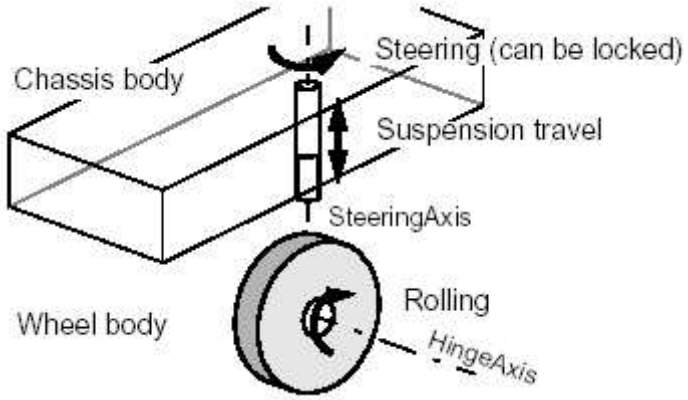
Le implementazioni dei robot di UsarSim derivano dal modello astratto *KRobot*, a sua volta derivato dalla classe *KVehicle* (descritta nel Karma Engine), che caratterizza precisamente la cinematica di un veicolo.

Il modello astratto *KRobot* descrive lo chassis, le parti (ruote, collegamenti, frame della telecamera ecc.) ed altri oggetti ausiliari, quali sensori, faretti, ecc.

La connessione tra lo chassis e le parti viene operata tramite giunti simulati, controllabili remotamente. Ad esempio la connessione delle ruote viene realizzata tramite giunti di tipo *car-wheel joint*, in grado di connettere due parti tramite una coppia di assi.

Nell'esempio in Figura 2.6 sono evidenziati l'asse di spin (Hinge Axis), connesso alla ruota, e l'asse di sterzo e sospensione (Steering Axis), connesso allo chassis.

Figura 2.6: Modellazione del sistema ruota-giunto-asse



I comandi relativi al controllo dei giunti sono impartiti al robot attraverso messaggi nell'interfaccia Gamebots. Nel modello del robot sono specificati tre tipi di controllo dei giunti:

- il controllo di *ordine zero (zero-order)* fa ruotare il giunto di un angolo specificato;
- il controllo di *ordine uno (first-order)* fa ruotare il giunto con una determinata velocità di spin;
- il controllo di *ordine due (second-order)* applica uno specificato momento di torsione al giunto.

E' opportuno notare che i tre tipi di controllo descritti possono essere applicati a qualsiasi giunto del robot, quindi anche ai giunti che realizzano l'eventuale sistema pan-tilt a supporto della telecamera.

Utilizzando il modello astratto *KRobot* come base di partenza, è possibile costruire una riproduzione del proprio robot, programmando in codice *Unreal Script*.

2.3.3 I Sensori

I Sensori sono modellati in UsarSim come Attori, con la limitazione di non poter essere inseriti direttamente nella simulazione, ma solo come componenti di un robot.

Ogni sensore può essere semplicemente montato sul robot aggiungendo una riga di codice sul file di testo relativo alla configurazione del simulatore. Devono essere specificati il nome del sensore, il tipo e la posa (posizione ed orientamento) in cui è montato.

Possono essere elencate ulteriori proprietà specifiche del tipo di sensore, ad esempio la massima distanza raggiungibile dal laser scanner o il campo di vista della telecamera; se queste non vengono specificate, UsarSim utilizzerà dei valori di default, anch'essi modificabili nel file di configurazione. La visualizzazione dei sensori nella simulazione può essere attivata e disattivata agendo sul parametro booleano *HiddenSensor*.

I sensori attualmente modellati in UsarSim sono:

Range Sensor. E' il sensore più semplice, modellato tramite la propagazione di un raggio lungo una direzione; fornisce la distanza del primo oggetto incontrato.

Infrarosso. Simile al *Range Sensor*, tale sensore è modellato tramite la propagazione di un raggio elettromagnetico di radiazione infrarossa, avente tra l'altro la proprietà di attraversare gli oggetti trasparenti.

Sonar. Il sensore Sonar fornisce la distanza dell'ostacolo più vicino all'interno di un cono d'azione identificato dalla posa del Sonar e dall'angolo di apertura del cono.

Laser scanner. Il laser scanner è modellato come una schiera di sensori infrarossi sistemati nello stesso punto (il centro del laser), con differenti orientazioni (generalmente 180 raggi a coprire un angolo piano). Dal sensore infrarosso il laser scanner eredita le proprietà dei raggi, tra cui quella di attraversare gli oggetti trasparenti.

Encoder. L'encoder fornisce l'informazione sull'angolo di rotazione di un giunto.

Odometria. Il sensore attuale fornisce una stima della posizione e dell'orientamento del robot a livello planare; i dati generati sono quindi una tripla x,y,theta; il sensore è strutturato in due parti:

1. una trasformazione di coordinate tra il sistema di riferimento assoluto del simulatore e quello centrato sul robot;
2. un calcolo dell'errore che simula l'errore del sensore di odometria;

Unità inerziale. Il sensore fornisce un'informazione relativa all'orientamento del robot.

Telecamera. E' il sensore più importante di UsarSim, nonché l'unico che fornisca dati esternamente al protocollo *GameBots*, in quanto l'output grafico è disegnato direttamente sulla finestra dell'*Unreal Client*. L'utilizzo dei dati della telecamera è quindi subordinato alla cattura dell'output video dell'*Unreal Client*.
Per la piattaforma MS Windows è distribuito con UsarSim il programma *ImageServer*, il quale, selezionata un'istanza di *Unreal Client* in esecuzione, ne distribuisce in rete il contenuto (via TCP/IP) ai client connessi, utilizzando un protocollo ad-hoc. Le opzioni di configurazione di *ImageServer* sono mostrate nella Tabella 2.2 .

Sistema pan-tilt. Tramite tale sistema, utilizzato in genere come supporto ad una o più telecamere, è possibile ruotare gli oggetti agganciati lungo l'asse orizzontale (pan) e quello verticale (tilt).

Human motion detection. Tale sensore simula un sistema piroelettrico, sensibile al calore, in grado di fornire la probabilità di aver rilevato il movimento di una vittima nella scena, all'interno del suo cono di visione.

Tabella 2.2: Opzioni di *ImageServer*

Opzione	**Descrizione**	**Default**
Comando per Unreal Client	Comando utilizzato per lanciare *Unreal Client*	`path` di installazione di Unreal
Risoluzione	Risoluzione video delle immagini trasmesse	320x240
Formato immagini	Formato delle immagini, utilizzando una trasmissione seriale (*raw*) dei pixel oppure il formato Jpeg a differenti livelli di compressione	jpg, qualità media
Frame rate	Frame rate massimo (il frame rate attuale viene modificato dinamicamente a seconda delle capacità del server)	10 fps
Numero di porta	Numero di porta del socket utilizzato per trasmettere le immagini	3000

Sensore di suono. Il sensore fornisce informazioni su ogni suono (urti, sirene, tonfi, grida d'aiuto delle vittime) rilevato nell'ambiente circostante il robot, in termini di volume e durata. Il volume rilevato decresce col quadrato della distanza dalla sorgente del suono.

Sensori di stato. Tramite i sensori di stato vengono forniti dati relativi al robot, indipendenti dall'ambiente esterno (propriocettivi), quali il livello della batteria di alimentazione del robot e lo stato di accensione dei faretti.

Tutti i sensori ereditano dalla classe astratta Sensor. Le specifiche dei sensori ed il loro comportamento (input accettati, output generati) vengono descritti in Unreal Script, il linguaggio di scripting interno al sistema. Come esempio della descrizione di un sensore si può leggere nell'appendice 2.4 il codice del Range Scanner Sensor, che modella uno scanner laser.

I sensori di distanza ereditano dalla classe Range Sensor, che fornisce la funzione per determinare la distanza di un corpo; tali sensori utilizzano

il parametro *MaxRange*, che permette di configurare la massima distanza raggiungibile.

In UsarSim è utilizzata una particolare unità di misura per gli angoli e le distanze, denominata *Unreal Unit* (UU), ereditata dall'*Unreal Engine*. Per le distanze vale l'equivalenza $1UU = 4mm$; per gli angoli vale l'equivalenza $65535UU = 360°$, da cui si ricava $1UU \simeq 0,00549°$. Analogamente, per le velocità lineari vengono utilizzati UU/s.

Tutti i sensori, con l'esclusione dei sensori di stato e della telecamera, offrono la possibilità di aggiungere rumore ed applicare distorsioni ai dati forniti, configurando i seguenti parametri:

- **Noise**, l'ampiezza relativa del disturbo casuale (random noise); col disturbo, il dato in output sarà $dataOutput = (1 + randomNoise) * dataLettura$;
- **OutputCurve**, la curva di distorsione, descritta tramite l'insieme dei punti che la compongono.

2.4 Vantaggi, limiti, prestazioni e miglioramenti possibili

In questa sezione verrà operata una disamina degli attuali vantaggi e limiti derivati dall'utilizzo di UsarSim come ambiente di simulazione per lo sviluppo di software orientato al soccorso robotico. In secondo luogo verranno discusse le prestazioni del simulatore. Chiuderà la sezione un paragrafo di proposte per miglioramento del simulatore, che delineano la situazione futura di un software il cui sviluppo procede tuttora a pieno ritmo.

2.4.1 Vantaggi

UsarSim presenta tutti i noti vantaggi di una simulazione 3D ad alta fedeltà:

- accurata ricostruzione della meccanica del robot, in termini di ruote, assi e giunti;
- disponibilità di differenti materiali, ad esempio vetri e specchi;
- simulazione di interazioni complesse con l'ambiente, inclusi ostacoli mobili e macerie;
- modellazione realistica di sorgenti di luce e condizioni atmosferiche, come pioggia o nebbia.

Altri vantaggi di UsarSim derivano dalla sua natura orientata al gioco online:

- la disponibilità per tutte le principali piattaforme software in commercio (Linux, Mac Os X e MS Windows);
- il semplice protocollo di connessione/comunicazione, basato su stringhe di testo trasmesse via TCP/IP;
- il supporto naturale alle simulazioni multiagente, con un numero di agenti anche elevato;
- il costo relativamente contenuto dell'*Unreal Engine*, dovuto alla forte concorrenza nel campo dei giochi FPS;
- l'utilizzo dei più avanzati processori grafici disponibili su pc a basso costo (Nvidia, ATI);
- la disponibilità di sofisticati ambienti IDE[15] per la modellazione di attori ed ambienti.

UsarSim supporta gli studi sull'*interazione uomo-macchina* (human-robot interaction, HRI) riproducendo accuratamente gli elementi dell'interfaccia utente (in particolare la videocamera), l'automazione ed il comportamento dei robot e l'ambiente remoto che collega la consapevolezza dell'operatore coi comportamenti dei robot.

[15] IDE: Integrated Development Environment, Ambienti integrati di sviluppo.

2.4.2 Limiti

E' possibile ravvisare alcune limitazioni nell'utilizzo del simulatore nel contesto Rescue, precisamente:

- non è possibile accedere direttamente al codice sorgente del motore grafico e del motore fisico dell'*Unreal Engine* (problema derivato dalla natura commerciale del prodotto); di conseguenza non è possibile operare modifiche ed ottimizzazioni a basso livello sul simulatore;
- il feedback video è limitato ad un solo Osservatore per ogni macchina (sistema operativo) impiegata; più precisamente, non è possibile avere istanze multiple dell'*Unreal Client* attive sulla stessa macchina;
- alcuni sensori del mondo reale relativi a concetti non implementati nel mondo di *Unreal*, quali ad esempio l'anidride carbonica, non sono utilizzabili.

2.4.3 Prestazioni

Il simulatore richiede un hardware orientato all'ambiente videoludico; i requisiti della versione di UsarSim derivata da UT2003 sono i seguenti:

- almeno 1 GHz di frequenza di clock;
- almeno 3 GB di spazio sull'hard disk;
- una scheda grafica con accelerazione 3D compatibile con OpenGL.

Nel rispetto di questi requisiti, il server UsarSim richiede poche risorse di sistema. Il server, senza modifiche, è in grado di accettare fino a 32 robot forniti di client spettatore.

L'*Unreal Client* (spettatore, con feedback video), al contrario, richiede più risorse; in ambiente Windows, il programma occupa circa 150 MB di memoria RAM.

2.4.4 Miglioramenti possibili

UsarSim è un simulatore estremamente giovane e costantemente sotto sviluppo da parte di una comunità eterogenea. Riporto in un breve elenco i punti focali su cui è incentrato lo sviluppo della prossima release.

Dal punto di vista tecnico:

- eventi dinamici, quali ad esempio collassi strutturali;
- simulazione di eventi ambientali, quali fuoco, acqua, fumo;
- simulazione di problematiche nella connessione radio.

Dal punto di vista organizzativo:

- trasferimento verso un repository opensource, quale Sourceforge;
- trasferimento al NIST dell'amministrazione, come per le arene fisiche;
- numerazione delle versioni.

Capitolo 3

Presentazione della piattaforma di sviluppo Spqr-Rdk

Il mio lavoro è stato svolto presso il laboratorio SIED[1] (Sistemi intelligenti per le Emergenze e la Difesa civile), nato dalla collaborazione tra l'Istituto Superiore Antincendi ed il DIS[2] (Dipartimento di Informatica e Sistemistica) "Antonio Ruperti" dell'Università degli studi di Roma "La Sapienza", con l'obiettivo di svolgere attività di ricerca volte allo sviluppo di metodologie, tecniche e strumenti prototipali da utilizzare in Operazioni di Soccorso.

All'interno del DIS è stata creata ed estesa in vari anni di lavoro una piattaforma di sviluppo modulare denominata Spqr-Rdk[3] (Software Per Qualunque Robot - Robot Development Kit), presentata in [9, 10] e disponibile per i sistemi operativi Linux e Mac OS X.

L'Spqr-Rdk è composto da un insieme di librerie software, driver di basso livello, moduli per comportamenti ad alto livello, interfacce verso gli agenti robotici ed utility grafiche per l'operatore.

Inizialmente l'Spqr-Rdk è stato sviluppato per essere utilizzato nelle

[1]SIED, Sistemi Intelligenti per le Emergenze e la Difesa civile `http://www.dis.uniroma1.it/~multirob/sied`

[2]DIS, Dipartimento di Informatica e Sistemistica `http://www.dis.uniroma1.it`

[3]Spqr-Rdk `http://www.dis.uniroma1.it/~spqr/Rescue.htm`

competizioni della *RoboCup Soccer*; successivamente le sue funzionalità sono state estese ed attualmente viene utilizzato in vari ambienti e con tipi diversi di robot (dai *Pioneer* della *ActiveMedia Robotics* ai *Sony Aibo*) e dispositivi di input (sonar, laser, stereovisione, sensori a infrarosso).

Tramite la piattaforma Spqr-Rdk è possibile dialogare con un agente robotico (reale o virtuale), osservarne lo stato, inviare istruzioni di movimento o fargli eseguire alcuni compiti di interesse quali localizzazione, mapping, riconoscimento di persone, navigazione autonoma.

I moduli software all'interno dell'Spqr-Rdk di maggiore interesse nell'ambito di questa tesi sono `ragent`, che permette di controllare un robot attraverso logiche decisionali di alto livello, e `rconsole`, un'interfaccia grafica che permette di monitorare e modificare lo stato di robot controllato da `ragent`.

3.1 Logiche decisionali in Spqr-Rdk: i Task

L'astrazione delle logiche decisionali e di controllo è modellata nel modulo `ragent` attraverso il concetto di Task. Ogni Task rappresenta un'entità decisionale autonoma dedita ad un compito specifico.

Esempi di Task sono il *SimpleMappingTask*, che costruisce incrementalmente la mappa dell'ambiente basandosi sulle letture di uno scanner laser, ed il *PanTiltControlTask*, che permette di controllare da remoto i movimenti di un sistema pan-tilt.

Ogni Task viene eseguito in un thread schedulato ciclicamente in parallelo agli altri. I Task sono in grado di scambiare dati tra loro, sia all'interno dello stesso programma sia tra istanze diverse di `ragent`; quest'ultima modalità è utilizzata nei contesti multiagente o quando sia necessario utilizzare un sistema di calcolo distribuito.

3.2 Astrazione del robot in Spqr-Rdk: il RobotTask

I *RobotTask* sono i Task che modellano i robot ed i sensori, realizzandone un'astrazione ad alto livello. Ogni istanza di `ragent` necessita la presenza di uno ed un solo *RobotTask*, che fornisce un'interfaccia standard verso un robot generico.

Il *RobotTask* gestisce in maniera autonoma il dialogo coi componenti di basso livello, cioè l'hardware nel caso di robot reali oppure strutture software nel caso di robot simulati. In questo modo tutti i Task dell'Spqr-Rdk, dialogando col *RobotTask*, sono in grado di astrarre dallo specifico robot utilizzato. Il dialogo è realizzato attraverso il meccanismo delle Code.

3.3 Flussi di dati in Spqr-Rdk: le Code

All'interno della piattaforma Spqr-Rdk il processo robotico viene astratto tramite un modello incentrato sui flussi di dati scambiati tra la parte sensoriale/attuativa e la parte decisionale, oppure all'interno della parte decisionale.

Il robot viene quindi rappresentato secondo un modello funzionale, rinunciando alla ricostruzione gerarchica degli oggetti meccanici, elettronici ed informatici in gioco e concentrando l'attenzione sul dialogo interno che essi realizzano.

Il coordinamento delle attività esige una considerazione attenta dell'affidabilità dei dati utilizzati, il cui valore viene legato all'intorno temporale del rilevamento e dalla relazione col contesto di informazioni eterogenee acquisite nello stesso intorno.

In questo senso, l'astrazione funzionale incentrata sui flussi di dati, completamente asincroni tra loro, necessita una gestione accurata della tempistica nel processo robotico: ad ogni informazione incapsulata nel flusso

viene associato il tempo di rilevamento, detto *timestamp*.

I flussi di dati sono gestiti tramite il meccanismo delle Code. Ogni Coda mantiene le informazioni scritte dal proprietario distribuendole a tutti i Task (lettori) interessati in maniera indipendente; in questo modo l'eventuale accumulazione dei dati può essere gestita in maniera indipendente da ciascun Task, secondo logiche di interpolazione e verifica o attraverso il semplice scarto delle informazioni obsolete.

Il *RobotTask* è incaricato della gestione dei dati provenienti dai sensori, che incapsula in oggetti e distribuisce all'interno di una o più Code. Più in generale, ogni Task può gestire una o più Code per distribuire i dati che processa. I comandi impartiti al robot (movimentazione, rotazione della telecamera) vengono incapsulati a loro volta in una Coda, con la sola differenza che in questo caso la Coda permette l'accesso in scrittura a tutti i Task interessati e l'accesso in lettura al solo *RobotTask*, che discrimina i comandi e gestisce la loro attuazione.

3.4 I principali *Sottosistemi* dell'Spqr-Rdk

I *Sottosistemi* sono utilizzati nell'Spqr-Rdk per modellare le logiche decisionali più complesse. Ogni *Sottosistema* è costituito da una gerarchia coordinata di Task ad alta coesione.

Nel seguito descriverò sommariamente i principali *Sottosistemi* dell'Spqr-Rdk, che ho utilizzato e modificato in tutto l'arco del mio lavoro di tesi.

3.4.1 Il *Sottosistema di Localizzazione e Mapping*

Il *Sottosistema di Localizzazione e Mapping* si occupa di stimare consistentemente la posizione del robot e la mappa dell'ambiente, a partire dalla storia sensoriale:

$$p(m_t, x_t | z_{0:t}, u_{0:t})$$

A tale scopo il *Sottosistema* coordina vari Task, tra i quali i più importanti sono:

- il *Task di Scan Matching*, nel quale viene operato l'allineamento delle letture dei sensori allo scopo di determinare la posa attuale del robot;
- il *Task di Mapping*, nel quale viene mantenuto lo stato della mappa;
- il *Task di Rilevamento Vetri*, tramite il quale viene operato il tracciamento delle superfici trasparenti.

L'idea di base è quella di utilizzare un solo sensore affidabile (ma incapace di rilevare alcuni tipi di superfici), quale lo scanner laser, per localizzare il robot, e di sfruttare i sensori ad ultrasuoni, quali i sonar, nella successiva fase di mapping.

3.4.1.1 Il *Task di Scan Matching*

il *Task di Scan Matching* utilizza una tecnica di *Maximum Likelihood* incrementale, allo scopo di ottenere ad ogni istante la posa del robot che massimizza la verosimiglianza dell'osservazione corrente z_t. Tale processo puo essere espresso analiticamente come

$$x_t^* = argmax_{x_t} p(x_t | m_{t-1}, x_{t-1}, z_t, u_t)$$

dove x_{t-1} indica la migliore stima della posa del robot all'istante precedente, m_{t-1} indica la mappa costruita fino all'istante precedente ed u_t la lettura proveniente dai sensori propriocettivi (encoder).

Ad ogni passo viene considerata la mappa costruita al passo precedente, alla quale viene applicato un filtro gaussiano per ridurre il rumore proveniente dai sensori.

La procedura quindi considera i dati provenienti dall'osservazione corrente, che vengono ruotati e traslati sulla mappa ottenuta dal filtraggio, per determinare la roto-traslazione che massimizza la sovrapposizione tra dati e mappa; la posa in cui si ha la sovrapposizione massima è la nuova posa.

Il *Task di Scan Matching* utilizza ulteriori tecniche di consolidamento della mappa, descritte in [12].

3.4.1.2 Il *Task di Mapping e Rilevamento dei Vetri*

Lo stato della mappa dell'ambiente è mantenuto dal *Task di Mapping e Rilevamento dei Vetri*, basato sul lavoro presentato in [13]. Il metodo utilizzato ha l'obiettivo di ottenere una mappa completa con tutti gli ostacoli presenti nel mondo, ma utilizzando per ognuno di essi solo l'informazione del sensore più preciso per quel particolare tipo di oggetto fisico.

Attualmente il metodo viene applicato utilizzando i dati provenienti da uno scanner laser e da un anello di sonar. Nell'ipotesi di ostacoli costituiti da materiali eterogenei, vengono distinti gli oggetti opachi (rilevabili dal laser e dal sonar) dagli oggetti trasparenti (rilevabili dal sonar).

Nell'ipotesi che ogni cella della mappa finale sia visibile dal robot con almeno uno dei sensori di cui dispone, le letture dei sensori sono memorizzate separatamente, tramite rappresentazioni metriche, dette *occupancy grid*, basate su una griglia a due dimensioni, in cui ogni cella $m_{x,y}$ contiene la probabilità che la corrispondente regione nell'ambiente sia occupata o libera. Per ogni sensore viene memorizzata anche una mappa privata a tre valori (libero, occupato, sconosciuto), ricavata a partire dalla propria *occupancy grid*.

Nel processo di aggiornamento sono considerate solo le celle di ogni *occupancy grid* coinvolte nelle ultime letture dei sensori. Ad ogni iterazione vengono aggiornate le probabilità di occupazione di ogni cella in base al valore attuale, alle letture ricevute ed ai valori presenti nelle celle adiacenti.

L'aggiornamento della mappa finale è ottenuto in base ad una serie di regole applicate ai tre stati possibili assunti da una cella nelle mappe private di ogni sensore:

- dove non si hanno informazioni del laser non viene eseguito alcun aggiornamento;

- dove si hanno solamente informazioni del laser vengono aggiornate le relative celle;
- dove si hanno informazioni sia del laser che dei sonar viene utilizzato il *metodo di integrazione sensoriale* per decidere se una cella sia occupata o libera.

Il *metodo di integrazione sensoriale*, utilizzato nel terzo caso, consiste nel cercare di motivare le mappe private ottenute dalle letture dei sensori comparandone le informazioni. Gli ostacoli rilevati da entrambi i sensori sono considerati ostacoli opachi; gli ostacoli rilevati dal sonar, per i quali non sono trovate corrispondenze nella mappa del laser, sono considerati ostacoli trasparenti.

3.4.2 Il *Sottosistema di Esplorazione Autonoma*

Il *Sottosistema di Esplorazione Autonoma* coordina vari Task allo scopo di ottimizzare l'esecuzione parallela dei compiti di esplorazione dell'ambiente ed identificazione delle vittime.

La strategia implementata nel *Sottosistema* può essere riassunta nei seguenti punti:

1. calcolo delle frontiere inesplorate;
2. recupero delle posizioni in cui sono state viste delle vittime;
3. esecuzione dell'algoritmo di selezione allo scopo di determinare la posizione ottimale;
4. navigazione verso la posizione ottima, mentre ad intervalli regolari si controlla se nella vicinanza sono state individuate vittime da esaminare;
5. raggiunta la destinazione, nel caso in cui si tratti di una possibile vittima, attivazione del modulo per l'analisi approfondita delle vittime.

L'algoritmo di scelta del prossimo obiettivo dell'esplorazione sfrutta l'ottimizzazione multiobiettivo, allo scopo di minimizzare il tempo di esplorazione e contemporaneamente ottenere il massimo dell'informazione possibile dall'ambiente.

3.4.3 Il *Sottosistema di Pianificazione del Moto*

Nel contesto USAR il *path-planning* non deve aver a che fare con ostacoli in movimento (a meno che non siano presenti altri agenti robotici), ma con una mappa che viene costruita man mano che prosegue l'esplorazione.

Di conseguenza il *Sottosistema di Pianificazione del Moto* è stato organizzato in una gerarchia, in cui il Task *Controllore della Pianificazione* coordina l'esecuzione di due Task:

- il *Pianificatore Topologico*, col compito di gestire i piani ad alto livello, adattandoli all'evoluzione della mappa;
- il *Motion Planner*, che si occupa di muovere il robot a basso livello, in una porzione ristretta della mappa.

Ogni volta che il *Controllore della Pianificazione* viene attivato con l'obiettivo di far raggiungere al robot la destinazione prescelta, esso richiede al *Pianificatore Topologico* il calcolo di un percorso ad alto livello e ne segue ogni passo dell'esecuzione tramite il *Motion Planner*.

In caso di fallimento del *Motion Planner* nel raggiungere una tappa intermedia, il *Controllore* riattiva nuovamente il *Pianificatore Topologico* richiedendo un nuovo percorso, da eseguire passo passo fino al raggiungimento della destinazione o all'esaurimento dei percorsi disponibili.

3.4.3.1 Il *Pianificatore Topologico*

Il *Pianificatore Topologico* utilizza l'algoritmo DPTM (Dynamic Probabilistic Topological Map), descritto in [14, 21]. Tale algoritmo si basa sul metodo della PRM (Probabilistic RoadMap), presentata in [19, 20], e la adatta

ad un ambiente sconosciuto, la cui mappa viene continuamente aggiornata ed espansa durante l'esecuzione.

L'algoritmo *DPTM* consiste nello scegliere posizioni casuali nello *spazio delle configurazioni* e nel tentare di collegarle tra loro, dove per collegate si intendono due posizioni tra le quali cui sia possibile seguire un cammino lineare, senza urtare alcun ostacolo.

Scelta una posizione casuale, essa viene aggiunta alla DPTM solo nel caso in cui:

- la posizione non può "essere vista" da nessun nodo della DPTM;
- l'inserimento della posizione nella DPTM ha come conseguenza di collegare due nodi già nella rete.

L'algoritmo si adatta ai cambiamenti della mappa tramite il GNG (Growing Neural Gas), una rete neurale ad apprendimento non supervisionato, utilizzata allo scopo di ridurre le dimensioni dello *spazio di input*. Se una posizione non può essere aggiunta alla *DPTM*, viene utilizzato il GNG per ridurre l'errore di distorsione, spostando leggermente la posizione nel tentativo di renderla accettabile alla *DPTM*.

Riguardo agli archi, l'algoritmo cerca di mantenerne un numero minimo; le ridondanze sono ridotte tramite il GNG, che si limita a collegare ogni nuovo nodo solamente ai due nodi più vicini; nel caso il nodo renda possibile il collegamento di due nodi già nella rete, vengono inseriti anche i due archi relativi a tale condizione.

La presenza di un numero limitato di archi e nodi rende possibile adattare velocemente la mappa ai cambiamenti dell'ambiente, ad esempio nell'eliminazione di un arco che *Task di Controllo della Pianificazione*, su notifica del *Motion Planner*, dichiara irraggiungibile.

3.4.3.2 Il *Motion Planner*

Il *Motion Planner* si occupa di pianificare ed eseguire una traiettoria di moto in una porzione ristretta della mappa.

L'algoritmo del *Motion Planner* (presentato in [21]) è basato sul Randomized Kinodynamic Planner (descritto in [22]), il quale a sua volta è un'estensione del noto Rapid-exploring Random Tree (presentato in [23]).

Il *Motion Planner* genera un *albero delle pose* che viene esteso dal punto di partenza verso la destinazione del moto. Ad ogni passo viene individuato un nodo da cui espandere l'albero e viene generato casualmente un set di candidati tra i quali selezionare il nodo successore da aggiungere all'albero.

La selezione del nodo successore è operata in base alla distanza nello spazio delle pose[4], calcolata per ogni candidato rispetto alla destinazione da raggiungere: viene selezionato il nodo a distanza minore, nel rispetto del vincolo che nella posa indicata dal nodo non sia presente alcun ostacolo.

L'esecuzione del piano durante il processo di generazione dell'*albero delle pose* stesso fornisce al robot un'alta responsività: l'algoritmo tende a generare piccoli piani di rapida attuazione, il cui raffinamento è operato durante la navigazione. Ad ogni passo dell'esecuzione del piano, l'albero di pose viene roto-traslato in accordo alla nuova posizione raggiunta dal robot, così da mantenere la consistenza delle informazioni e consentire un rapido adattamento nella generazione delle posizioni successive.

3.4.4 Il *Sottosistema di Riconoscimento delle Vittime*

Il riconoscimento del corpo umano è uno dei temi di ricerca più studiato negli ultimi anni. In generale l'ambito del riconoscimento degli oggetti può essere diviso in due sotto-gruppi: il riconoscimento degli oggetti rigidi ed il riconoscimento degli oggetti deformabili.

Considerando il corpo umano come una forma composta dai diversi elementi deformabili con un altissimo grado di libertà è preferibile utilizzare metodi ibridi in grado di riconoscere le parti rigide e quelle deformabili.

Il metodo utilizzato nel *Sottosistema di Riconoscimento delle Vittime* è composto dalle seguenti fasi:

[4]La distanza nello spazio delle pose (x, y, θ) viene calcolata con la formula $dist = \sqrt{(\triangle x)^2 + (\triangle y)^2 + k(\triangle \theta)^2}$, in cui k è un fattore di peso che misura la rilevanza dell'orientamento del robot.

- riconoscimento dei bordi;
- chiusura dei contorni;
- calcolo dello spessore dei contorni, tramite la ricostruzione della terza dimensione usando tecniche di Stereovisione;
- associazione delle forme ricavate dall'immagine stereo ad una base di conoscenza, usando un filtro di somiglianza Bayesiana.

Nella prima fase del processo di riconoscimento vengono ricavati i bordi degli oggetti presenti nello scenario, tramite l'utilizzo di un filtro di colore YUV. L'algoritmo calcola per ogni coppia delle immagini una soglia basata sull'entropia dei colori negli oggetti.

Una volta individuata i bordi, viene adottata una strategia di tipo *Splitting and Merging* per chiudere i contorni, in maniera tale da conservare solo i contorni completi e escludere i rumori e i contorni che sono allocati dentro i contorni chiusi.

L'algoritmo nella fase di *Splitting* cerca di chiudere tutti i contorni usando i metodi di Gap-Filling ed in seguito elimina tutti i contorni che non è riuscito a chiudere. Nella fase di *Merging* vengono eliminati, tra i contorni rimasti, quelli collocati all'interno di contorni chiusi più grandi.

Nella fase di ricostruzione della terza dimensione viene calcolato lo spessore dei contorni chiusi, tramite tecniche di stereovisione. Considerando che il metodo è basato sulle forme, in questa fase si cerca di rendere il metodo insensibile ai problemi connessi al cambiamento dei punti di vista ed in generale alla proiezione delle immagini bidimensionali in uno spazio tridimensionale.

Per poter calcolare lo spessore dei contorni vengono scelti alcuni punti pesati all'interno d'ogni contorno; lo spessore del contorno è calcolato tramite la media pesata di quei punti.

Nell'ultima fase i contorni ricavati sono confrontati con la base di conoscenza dell'Spqr-Rdk. Tale base è stata costruita usando diversi modelli del corpo umano ed attualmente contiene più di 400 modelli stereo. La struttura della base di conoscenza poggia su 4 matrici e su una tabella delle relazioni tra i giunti del corpo.

Per poter effettuare il confronto è necessario dividere preliminarmente i contorni. Per ricavare i segmenti interni di ogni contorno viene applicata la regola delle minime di curvature; l'attendibilità di tali segmenti è controllata in seguito, utilizzando l'algoritmo di "Holfmann". L'ultimo passo è la stima della somiglianza tra i segmenti interni dei contorni e la base di conoscenza, ricavata attraverso una misura di somiglianza bayesiana.

Capitolo 4

Modellazione di uno scenario di soccorso in UsarSim

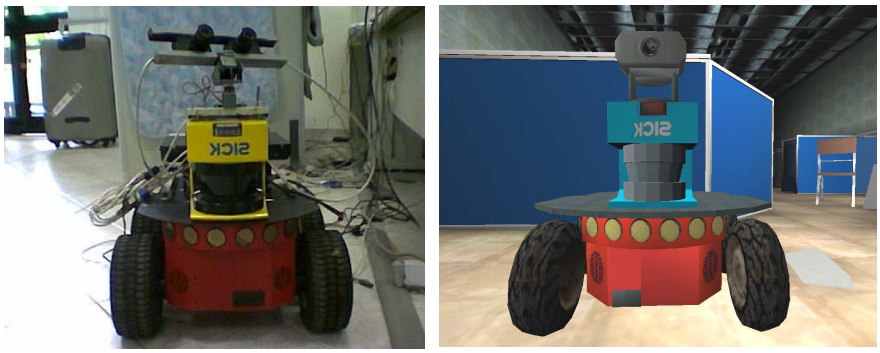
In questo capitolo viene descritto il lavoro svolto nella realizzazione dell'interfaccia tra il simulatore UsarSim, (presentato nel capitolo 2) e la piattaforma di sviluppo Spqr-Rdk (descritta nel capitolo 3), correntemente utilizzata nel laboratorio SIED.

Il lavoro di interfacciamento è stato svolto in parallelo sui due software. Inizialmente ho modellato in UsarSim il robot utilizzato nel laboratorio. In seguito ho dotato la piattaforma Spqr-Rdk di un insieme di driver verso il simulatore, allo scopo di virtualizzare completamente il robot e i suoi i sensori.

Successivamente ho esteso le potenzialità di UsarSim introducendo due nuovi sensori simulati, una Stereo Camera (coppia di telecamere sincronizzate) ed il prototipo di una Swiss Camera, un sensore in grado di ricostruire una scena 3D combinando informazioni di luminosità e distanza degli oggetti.

Infine ho eseguito dei test per validare l'interfaccia realizzata, utilizzando il *Task di Mapping e Rilevamento dei Vetri* disponibile in Spqr-Rdk.

Figura 4.1: Il nostro robot, in versione reale e simulata



4.1 Modellazione del robot in UsarSim

Nella prima fase della realizzazione dell'interfaccia ho provveduto a ricostruire in UsarSim una copia realistica del robot utilizzato in laboratorio.

4.1.1 Descrizione del robot

Il robot correntemente utilizzato nel laboratorio di ricerca SIED per lo sviluppo nel contesto USAR è mostrato nella Figura 4.1. Lo chassis (la base di colore rosso) è un modello su quattro ruote motrici Pioneer 3-AT (All Terrain) della *ActivMedia Robotics*, comprendente le ruote, i servomotori che le muovono, la scheda di interfaccia dei dati tra i sensori e il software, un anello di sonar circolare e la copertura in metallo.

Il Pioneer 3-AT è provvisto di un controllo che gli permette di ruotare su se stesso consentendo traiettorie con tratti a raggio di curvatura nullo (ovvero la cinematica del robot è a uniciclo). Le dimensioni dello chassis sono 50x24x26 cm; il diametro delle ruote è di 21,5 cm.

Sulla superficie superiore della base del robot è posto un laser scanner, modello PLS (proximity laser scanner) della casa produttrice Sick, che può effettuare scansioni su un semipiano posto di fronte a sé a intervalli di

mezzo grado o di un grado, con errore medio inferiore a pochi (<10) cm su distanze fino a 50 metri.

Sopra il laser è posto il sistema pan-tilt, una coppia di servomotori ad assi ortogonali costruito ad-hoc per sorreggere il peso della coppia di telecamere e permettere loro di compiere movimenti orizzontali (pan) e verticali (tilt).

Le telecamere stereo sono un modello costruito e commercializzato dall'istituto di ricerca SRI[1]. Al centro delle due telecamere è montato un sistema di misura della temperatura. Ai lati telle telecamere sono montati due faretti utilizzati per illuminare l'ambiente e facilitare il compito della stereovisione.

Sullo chassis del robot sono infine posti due calcolatori (un laptop ed un piccolo elaboratore progettato per usi industriali) collegati ai sensori e su cui viene svolta l'elaborazione software; gli elaboratori sono in comunicazione wireless con un eventuale operatore remoto che volesse inviare comandi al robot o solo osservarne lo stato.

4.1.2 Modellazione del robot

Lo chassis del robot P3AT è già modellato nella distribuzione standard di UsarSim. Ho configurato il robot simulato aggiungendo allo chassis virtuale tutti i sensori presenti nel robot reale.

E' possibile leggere la configurazione nel robot inserita nel file USAR.INI nell'appendice A.2. Nel testo vengono impostati alcuni parametri tra cui la massa dello chassis ed il tipo di sistema di riferimento della telecamera (assoluto o relativo); inoltre viene impostato l'inserimento delle ruote, del sistema pan-tilt, della telecamera, dei faretti, del sensore d'odometria, dei sedici sonar disposti ad anello ed infine del laser scanner.

[1]SRI - Stanford Research Institute `http://www.ai.sri.com/`

4.2 Realizzazione dell'interfaccia del robot e dei sensori principali

Una volta completata la modellazione del robot ho provveduto a realizzare l'interfaccia tra UsarSim e la piattaforma di sviluppo Spqr-Rdk. Nel seguito viene fornita una descrizione della piattaforma e del modo in cui è stata realizzata l'interfaccia col simulatore.

4.2.1 Robot e dati odometrici

Come mostrato nel paragrafo 3.1, all'interno della piattaforma di sviluppo Spqr-Rdk il generico robot viene modellato come un *RobotTask*. Seguendo questo approccio è stato creato in Spqr-Rdk lo *UsarRobotTask* allo scopo di contenere l'interfaccia di basso livello col simulatore UsarSim.

Lo *UsarRobotTask* task presenta una fase iniziale in cui viene inizializzato il collegamento al server UsarSim. I dati di connessione, tra cui l'ip e la porta del socket, sono salvabili e caricabili tramite file esterni di configurazione. Al termine di questa fase il robot viene inserito nell'arena.

In seguito il task entra in un ciclo infinito in cui ad ogni passo vengono prelevati e gestiti i dati dei sensori ed impartiti gli eventuali comandi al robot.

La gestione dei sensori viene operata ricorrendo alle Code, le strutture dati utilizzate nella piattaforma Spqr-Rdk presentate nel paragrafo 3.3. In quest'ottica ho realizzato in *UsarRobotTask* una generica Coda di letture, che gestisce tutti i dati forniti dai vari sensori del robot e li rende disponibili ai Task.

Il primo tipo di messaggi che ho provveduto ad interfacciare sono stati quelli contenenti dati relativi alla posa del robot stimata dall'odometria, forniti in UsarSim dal sensore Odometry. Ho incapsulato tali dati in oggetti di tipo OdometryData e li ho inseriti all'interno della Coda di letture

generica di *UsarRobotTask*.

Al momento della realizzazione dell'interfaccia, nella versione di 0.1 di UsarSim (secondo la convenzione definita nel paragrafo 2.1.2), i dati forniti dal sensore Odometry erano privi di timestamp. Per ovviare a questo inconveniente ho provveduto a creare un sistema di temporizzazione indipendente dal server UsarSim, basato solamente sui tempi d'arrivo dei pacchetti all'interfaccia di Spqr-Rdk.

Nel frattempo ho sollevato il problema nella comunità di UsarSim, proponendo l'aggiunta di un timestamp in ogni messaggio del sensore. La proposta è stata accolta favorevolmente e dalla versione 0.2 i dati forniti da Odometry sono corredati da timestamp.

Attualmente, per quanto la temporizzazione dei dati sia affidata di default al simulatore, ho lasciato la possibilità di utilizzare una temporizzazione interna, indipendente dal server, basata solamente sui tempi di arrivo dei dati all'interfaccia di ingresso; il passaggio tra le due modalità può essere attivato da un flag nel file di configurazione di *UsarRobotTask* o direttamente da `rconsole`, l'interfaccia grafica di Spqr-Rdk.

Come ultimo passo ho implementato in *UsarRobotTask* un parser per i comandi inviati dai Task di controllo per la movimentazione del robot.

L'intercettazione di tali comandi avvia una fase di processamento dei dati, in cui viene operata la conversione dalle variabili in ingresso, relative alla velocità lineare ed angolare richiesta, a quelle in uscita, riconosciute da UsarSim, relative alla velocità di spin impressa sui motori sinistro e destro.

Al termine del processamento i dati sono inseriti in una stringa di comando ed inviati al simulatore.

4.2.2 Sonar e Scanner Laser

Nel rispetto del meccanismi di comunicazione utilizzati in Spqr-Rdk, ho realizzato l'interfaccia ai sensori Sonar e RangeScanner sfruttando la Coda

di letture gestita da *UsarRobotTask*. Ogni messaggio proveniente dai sensori, contenente i dati relativi ad uno scan completo di uno Scanner Laser o le letture di un set di Sonar, viene incapsulato in un oggetto di tipo, rispettivamente, LaserData o SonarData, ed inserito nella Coda.

Riguardo la temporizzazione, i sensori RangeScanner e Sonar presentavano in un primo momento lo stesso problema sollevato nel paragrafo precedente: la mancanza di timestamp. Dal dibattito sull'argomento nella comunità di UsarSim ho ottenuto che tale informazione fosse aggiunta anche nei dati di tali sensori.

4.2.3 Telecamera e pan-tilt-zoom

La telecamera è il sensore più in rilievo nell'ambiente simulato di UsarSim, utilizzata non solo come equipaggiamento del robot, ma anche per monitorare l'andamento della simulazione, grazie alla capacità di introspezione fornita dalle visuali esterne posizionabili ed orientabili in tempo reale.

Per la simulazione dell'output video della telecamera posta sul robot ho deciso di utilizzare il programma *ImageServer*, distribuito con UsarSim, descritto nella presente tesi a pagina 45.

Tale programma, esterno ad *Unreal Engine* e disponibile solamente per ambienti Windows, si occupa di catturare l'output video di una finestra *Unreal Client* e ne distribuisce il contenuto ai client connessi via TCP/IP secondo un protocollo ad-hoc.

Per realizzare l'interfaccia ho creato il nuovo Task *UsarVisionTask* all'interno di Spqr-Rdk, nel quale viene gestita la connessione ad *ImageServer*.

UsarVisionTask si occupa di memorizzare le immagini ricevute dal server in un oggetto di tipo RImage, reso disponibile al programma `rconsole` per la visualizzazione a schermo ed agli altri Task per le elaborazioni successive (ad esempio per il rilevamento dei contorni).

La connessione è gestita tramite un ciclo infinito in cui ad ogni passo dell'esecuzione del Task viene memorizzata un'immagine ed inviato al

server un comando di conferma. Di conseguenza, nei limiti del *frame rate* impostato su *ImageServer*, il tempo di aggiornamento dell'immagine è direttamente riconducibile al periodo di schedulamento di *UsarVisionTask*, configurabile all'avvio del programma `ragent`.

L'interfaccia al sistema pan-tilt-zoom è stata inserita all'interno di *UsarRobotTask*. Lo stato del sistema viene mantenuto tramite le tre proprietà `CameraPan`, `CameraTilt` e `CameraZoom`, accessibili da ogni Task ed aggiornate ad ogni lettura delle informazioni provenienti da UsarSim. Inoltre nel parser dei comandi di UsarRobotTask (implementato in 4.2.1) ho permesso l'intercettazione dei comandi di rotazione (pan e tilt) e zoom della Camera, che vengono processati (allo scopo di convertire le unità di misura) ed inoltrati al simulatore.

4.3 Realizzazione di nuovi sensori in UsarSim

Una volta interfacciati i sensori comuni tra UsarSim e l'Spqr-Rdk si è passati ad estendere UsarSim per ottenere le caratteristiche non ancora supportate. La priorità è stata assegnata alla Stereovisione, senza la quale non sarebbe stato possibile utilizzare in simulazione il *Sottosistema di Riconoscimento delle Vittime* dell'Spqr-Rdk. In un secondo momento si è pensato di creare un modello simulato della Swiss Camera, da testare in UsarSim in vista di un futuro acquisto del sensore reale.

4.3.1 UsarSim e Stereovisione

Nel costesto del soccorso robotico (USAR) uno degli obiettivi più importanti è il riconoscimento delle vittime, senza il quale rimangono vani tutti gli sforzi di localizzazione, mapping e navigazione.

Nell'Spqr-Rdk il *Sottosistema di Riconoscimento delle Vittime*, descritto nel paragrafo 3.4.4, utilizza tecniche basate sulla Stereovisione. Dopo

aver interfacciato tra loro i componenti base del nostro robot mi sono quindi occupato di emulare un sistema di telecamere stereo.

4.3.1.1 Un'amara sorpresa

Le tecniche di Stereovisione si basano sul confronto tra le immagini ottenute da due diversi punti di vista, allo scopo di ricostruire informazioni in tre dimensioni sull'ambiente osservato. È necessario che le sequenze video acquisite dalle telecamere siano sincronizzate, cioè che ogni coppia di immagini sia ottenuta nello stesso momento.

Nello sviluppo dell'interfaccia tra l'Spqr-Rdk e la versione 0.1 di UsarSim ho constatato l'impossibilità di realizzare una simulazione della Stereovisione: la causa è insita nell'architettura di Unreal, che permette l'esecuzione di una sola copia del programma *Unreal Client* nel sistema operativo. Questo significa che per quante telecamere siano montate nell'ambiente di simulazione, l'operatore può avere un solo feedback video (una sola finestra del programma) attivo per volta ed è costretto a ciclare tra le visualizzazioni disponibili.

Questa limitazione non era emersa in precedenza in quanto anche chi aveva dotato il robot di due telecamere o inserito più robot dotati di telecamera non era interessato ad avere due feedback video contemporaneamente. Tale problema deriva dalla natura di Unreal di gioco in prima persona e non è affrontabile direttamente, in quanto non è possibile conoscere e modificare il codice sorgente di Unreal.

Anche la prospettiva di utilizzare due diversi PC allo scopo di visualizzare copie distinte di *Unreal Client* non è risultata praticabile, in quanto non era possibile assicurare la sincronizzazione delle sequenze video delle telecamere.

D'altra parte, il nostro sistema di riconoscimento vittime necessitava per la Stereovisione di immagini catturate nello stesso istante. Tenendo a mente la limitazione di un solo feedback video, ho cercato un approccio

che utilizzasse l'area di una sola schermata per mandare le informazioni di entrambe le telecamere.

4.3.1.2 Implementazione della Stereovisione in UsarSim

La soluzione è stata implementata sulla versione simulata del robot utilizzato nel laboratorio, il cui lavoro di modellazione è stato mostrato in 4.1.2. Ho agito sul server UsarSim modificando la porzione di codice relativa alla definizione del robot P2AT, in codice Unreal Script; il codice può essere consultato nell'appendice A.1.

La modalità Stereovisione viene attivata tramite l'impostazione del parametro `useStereoVision` all'interno del file di configurazione USAR.INI di UsarSim, nella sezione relativa al robot utilizzato (un esempio di utilizzo è mostrato nell'appendice A.2).

La modifica principale ha riguardato la funzione `DrawHud`, in grado di disegnare in sovraimpressione sullo schermo. La funzione `DrawHud` viene utilizzata a tempo di esecuzione per simulare l'eventuale scaricamento definitivo delle batterie del robot, tramite l'oscuramento dello schermo e la comparsa di un banner recante la scritta "No Power".

La funzione `DrawHud` è stata ridefinita per reimpostare il feedback video dell'osservatore, cioè l'output grafico del programma *Unreal Client*. La sua precedente funzionalità è stata preservata richiamando al termine dell'esecuzione il codice della classe padre.

La finestra dell'*Unreal Client* è stata divisa in due semiquadri secondo la larghezza, in modo da accogliere l'immagine della telecamera sinistra nel semiquadro di sinistra e quella della telecamera di destra nel semiquadro di destra. Un esempio dell'output grafico ottenuto è mostrato nella Figura 4.2 .

La sincronizzazione delle immagini è assicurata in quanto esse vengono disegnate consecutivamente all'interno dello stesso frame.

Le immagini vengono tracciate tramite la funzione `DrawPortal`, che for-

Figura 4.2: Esempio di feedback video fornito da un sistema di telecamere stereo

nisce una vista sulla simulazione accettando come parametri il punto di vista (posizione ed orientamento) e l'ampiezza del campo di vista. I parametri utilizzati sono ereditati dalla configurazione della telecamera predefinita.

Alla posizione di vista viene sommato uno spiazzamento configurabile attraverso la variabile vettoriale `stereoSpacing`. Secondo le convezioni utilizzate in UsarSim, agendo sull'asse y si ottiene uno spiazzamento in larghezza, agendo sull'asse z in altezza e sull'asse x in profondità. La configurazione di default prevede uno spiazzamento sul solo asse y, dell'ampiezza di 5 cm, che produce un distanziamento tra le telecamere di 10 cm.

Il modello introdotto non prevede la convergenza del punto di visione delle due telecamere, che si limitano a guardare in parallelo. E' comunque possibile, all'occorrenza, modificare la convergenza con minime modifiche al codice.

La tecnica appena descritta presenta due apprezzabili vantaggi:

1. produce immagini stereo sincronizzate in un'unica schermata, semplificando l'ispezione visiva;

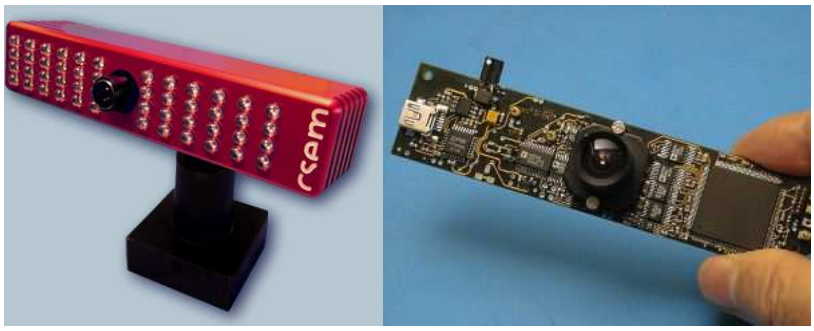
2. utilizza lo stesso canale di output (l'*Unreal Client*) del sensore Camera standard, semplificando l'acquisizione dei dati da parte dell'utente.

4.3.2 Uno sguardo nel futuro: la Swiss Ranger Camera

La Swiss Ranger Camera è un sensore in grado di ricostruire una scena 3D combinando informazioni di luminosità e distanza degli oggetti. Tali caratteristiche ne rendono apprezzabile l'utilizzo nel contesto USAR, in particolar modo nel'area di riconoscimento delle vittime.

Nel laboratorio SIED il dibattito sul dotarsi o meno di tale sensore è sfociato nella decisione di operare una serie di test in simulazione allo scopo di aiutare la valutazione degli effettivi costi e benefici (modalità proattiva di utilizzo del simulatore, come descritto nel paragrafo 1.3, a pagina 18). Considerate le caratteristiche del sensore, si è scelto di eseguire tali test

Figura 4.3: Swiss Ranger Camera, box esterno e scheda interna



utilizzando UsarSim. A questo scopo ho analizzato i dati tecnici della Swiss Ranger Camera e ne ho creato un modello simulato.

4.3.2.1 Descrizione del sensore

La Swiss Ranger Camera, mostrata a sinistra nella Figura 4.3, è un sistema ottico che offre immagini 3D ad alta risoluzione in tempo reale, realizzato dall'azienda svizzera CSEM[2]. La documentazione relativa alla SwissRanger Camera è stata reperita in [11].

Lo Swiss Ranger è un sistema standalone che racchiude all'interno di un unico box di dimensioni contenute (135x45x32 mm) un'unità di illuminazione, un sensore ottico 3D dedicato e componenti elettroniche di controllo.

Il sistema è basato sul principio *time-of-flight* (TOF) ed utilizza una sorgente di luce infrarossa modulata su frequenze vicine ai 20 MHz. Gli impulsi di luce emessi sono riflessi dagli oggetti nella scena. Quando gli impulsi tornano indietro alla Camera, l'elettronica inserita nel sensore demodula il segnale ricevuto per ogni pixel e calcola la sua fase, che è proporzionale alla distanza dell'oggetto riflettente. Inoltre per ogni pixel viene calcolata la luminosità dell'oggetto, utilizzando il ritardo del segnale ed una modulazione d'ampiezza. In questo modo il sensore è in grado di determinare sia una

[2]CSEM, Swiss Center for Electronics and Microtechnology `www.csem.ch`

mappa di luminosità che una mappa di distanza della scena, generando un modello 3D dell'ambiente. La risoluzione nella distanza misurata è inferiore ai 5 mm.

Le immagini di profondità e le convenzionali immagini in scala di grigi sono fornite attraverso un'interfaccia Usb, che viene utilizzata anche per configurare il sensore da remoto; le impostazioni sono memorizzate in una EEPROM interna.

La Camera è basata su un sensore *cmos/ccd*, mostrato a destra nella Figura 4.3, che offre una risoluzione massima di 124x160 pixel ed un *frame rate* massimo di 30 Hz ed una distanza massima di rilevazione di 7,5 metri.

4.3.2.2 Modellazione ed interfaccia del sensore

Dalle specifiche tecniche ho elaborato un modello delle proprietà del sensore simulato, in termini di risoluzione orizzontale e verticale, distanza massima, ampiezza del rumore e curva di rumore.

Ho scelto di modellare la Swiss Ranger Camera scindendo le informazioni di luminosità e distanza, fornite rispettivamente dal sensore Camera, già esistente, e dal nuovo sensore IRC, implementato ex novo.

Per fornire informazioni coerenti i due sensori devono essere montati nella stessa posizione e con lo stesso orientamento. In questo modo per ogni pixel dell'immagine fornita dalla Camera è disponibile la sua distanza dal punto di osservazione.

Ho scelto di chiamare il nuovo sensore Infrared Range Camera (IRC), in quanto basato sul rilevamento delle distanze tramite radiazione infrarossa. Ho inserito il sensore IRC nella gerarchia di UsarSim nel ramo dei misuratori di distanze (range detector), come sottoclasse del sensore InfraRed.

I dati forniti dalla IRC sono organizzati in una matrice rettangolare, le cui dimensioni sono la risoluzione orizzontale e verticale della videocamera. Purtroppo i test preliminari sul sensore hanno messo in evidenza un limite del sistema di comunicazione: il processamento e l'invio di una intera

immagine dell'IRC richiedevano tante e tali risorse di calcolo da impedire il corretto proseguimento della simulazione: la posizione del robot veniva aggiornata in ritardo, i dati dei sensori perdevano di coerenza.

Per porre rimedio al problema ho rimodellato il generico messaggio inviato dalla IRC come una lista di un numero arbitrario di scan (righe) orizzontali dell'immagine considerata, così da consentire maggiore flessibilità nell'invio dei dati; il numero di scan può essere configurato nel file USAR.INI.

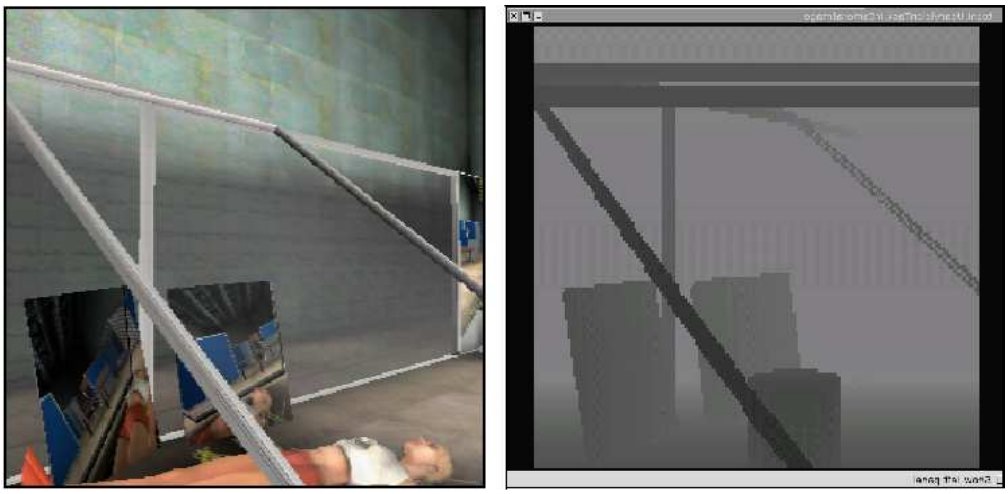
Ho inoltre ritenuto necessario proibire un utilizzo continuativo del sensore, reimplementandolo secondo il modello stimolo/risposta: ad ogni richiesta del client, l'IRC risponde inviando una serie di messaggi con gli scan orizzontali dell'immagine. Per garantire la coerenza dei dati il robot non deve eseguire alcun movimento fino alla ricezione dell'ultimo messaggio.

Per quanto riguarda la modellazione dell'interfaccia su Spqr-Rdk, ho inserito la comunicazione col sensore IRC nel Task *UsarVisionTask*, utilizzato in precedenza per interfacciare la telecamera, come descritto nel paragrafo 4.2.3.

La ricezione di una nuova immagine può essere abilitata interattivamente (dall'interfaccia grafica `rconsole`) oppure ad intervalli prestabiliti (programmabili attraverso Task). L'immagine viene memorizzata in un oggetto di tipo RImage, in modo analogo a quella fornita dalla telecamera, e visualizzata tramite il programma `rconsole`.

Un esempio del tipo di informazione fornita è mostrato nella Figura 4.4, in cui sono comparati il feedback della telecamera (a sinistra) e quello del sensore IRC (a destra), in cui la luminosità è proporzionale alla distanza; agli oggetti oltre la distanza massima rilevabile dal sensore viene associata la luminosità massima. In particolare, si può notare come il sensore non intercetti la superficie obliqua, a causa della sua trasparenza, ma solo il suo contorno. Le due bande orizzontali nella parte alta dell'immagine dell'IRC sono dovute ad un errore di sincronizzazione.

Figura 4.4: Immagine catturata dalla telecamera e corrispondente visuale fornita dal sensore IRC



Attualmente si pensa di utilizzare la Swiss Ranger Camera sul secondo robot, in sostituzione delle stereocamere. I dati forniti dalla Swiss Ranger Camera verranno utilizzati dal *Sottosistema di Riconoscimento delle Vittime*, in maniera simile ai dati forniti dalla stereovisione. L'unico cambiamento previsto al *Sottistema*, in questo constesto, sarà una diversa implementazione della fase di calcolo dello spessore dei contorni chiusi (descritta a pagina 61). Tale calcolo sarà eseguito tramite un processo di integrazione delle informazioni di luminosità e dei dati di distanza forniti dal sensore.

Come ultima considerazione, devo notare come l'attuale modello simulato della Swiss Ranger Camera sia in parte inadeguato. L'utilizzo del modello stimolo/risposta e l'acquisizione e l'invio dilazionato nel tempo dei dati sono resi necessari dalle limitazioni dell'attuale sistema di comunicazione di UsarSim, che nel mio lavoro ho contribuito a portare in evidenza. Questa pesante limitazione è dovuta infatti solamente alla necessità di preservare il corretto andamento della simulazione nel suo complesso.

Una simulazione più fedele sarà possibile, a mio avviso, solamente uti-

lizzando per il sensore IRC un diverso sistema di comunicazione, basato ad esempio su una connessione di rete dedicata, operante in parallelo alla connessione principale. In questo modo verrebbe garantita l'indipendenza dei flussi dati, senza pregiudicare l'andamento della simulazione. Inoltre sarebbe possibile adottare un protocollo di trasporto diverso per i dati del sensore IRC, svincolata dalla principale, passando ad esempio dal TCP al più leggero UDP.

Un metodo alternativo di comunicazione potrebbe riguardare l'inserimento dei dati di distanza, tramite un'opportuna codifica, all'interno dell'*Unreal Client*, nella stessa schermata che mostra i dati di luminosità.

4.4 Test di validazione dell'interfaccia

Al termine del lavoro di modellazione ed interfacciamento del robot e dei sensori ho ritenuto necessaria una verifica esaustiva del buon comportamento del sistema nel suo complesso.

Riflettendo sulla modalità più efficace di validazione, ho considerato l'esperienza già acquisita in questo senso da parte della comunità che sviluppa UsarSim. In questo contesto la maggior parte dei lavori si concentra sulla verifica dell'effettiva corrispondenza tra i sensori reali e le corrispettive versioni simulate; ad esempio i ricercatori dell'Università di Brema descrivono in [4] la validazione del sensore RangeScanner, mentre in [5] viene descritta la validazione della movimentazione di vari modelli di robot (PER, Corky, P2AT, P3AT) da parte dei team della Carneige Mellon University e dell'Università di Pittsburgh.

Dal mio canto, essendo interessato ad un controllo ad ampio spettro, ho deciso di concentrare l'attenzione sull'integrazione complessiva del robot e dei sensori, verificando il buon funzionamento di alcuni sottosistemi di SpqrRdk che implementassero logiche decisionali complesse basate su tecniche di fusione sensoriale.

A tal proposito, il test più significativo è stato quello del *Task di Map-*

ping e Rilevamento dei Vetri (abbreviato nel seguito con la sigla TMRV), un modulo del *Sottosistema di Localizzazione e Mapping*. Il TMRV, presentato nel paragrafo 3.4.1.2, utilizza e fonde i dati forniti dall'odometria, da uno scanner laser e da un anello di sonar per ricavare informazioni sulla presenza di ostacoli trasparenti.

4.4.1 Adeguamento preliminare del sensore RangeScanner

Per testare il TMRV su UsarSim ho predisposto un'arena simulata in cui accanto ai tipici pannelli verticali che delimitano gli ambienti sono posti alcuni pannelli di materiale trasparente.

Nel test preliminare è emerso un comportamento anomalo del sensore RangeScanner (versione simulata di uno scanner laser): i raggi del sensore anziché passare attraverso le superfici trasparenti, venivano intercettati e riflessi, come se le superfici fossero opache.

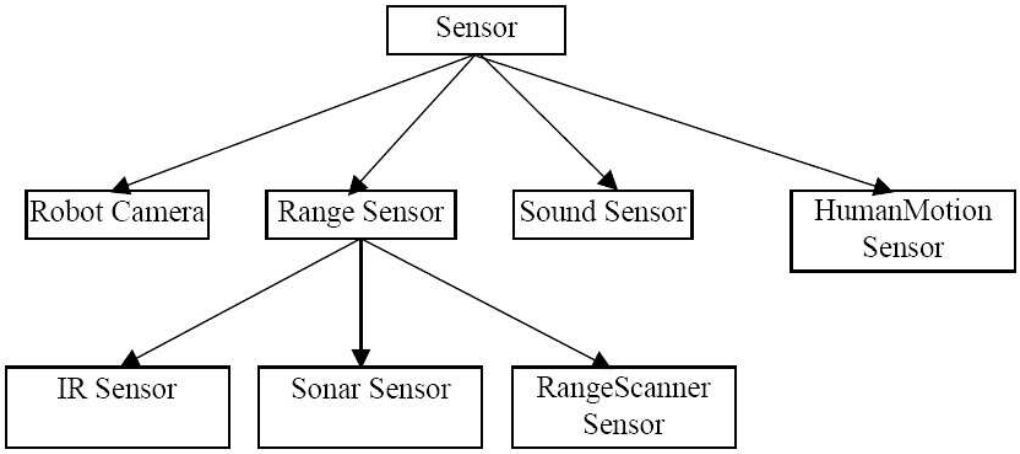
Di conseguenza, ho dovuto apportare delle modifiche al RangeScanner per evitare tale comportamento.

All'inizio della test del TMRV utilizzavo la release 0.1 di UsarSim, secondo la numerazione convenzionale delle versioni del simulatore definita in 2.1.2.

In questa release il sensore RangeScanner era inserito come sottoclasse del Range Sensor; il suo posto nella gerarchia (mostrata nella Figura 4.5) era giustificato dalla schematizzazione del RangeScanner come collezione di sensori di tipo Range Sensor; come risultato, il RangeScanner utilizzava la funzione `GetData` del Range Sensor per rilevare la distanza dell'ostacolo più vicino, per ogni raggio.

Al momento di apportare le modifiche al RangeScanner ho deciso di ricorrere al sensore IR (InfraRosso); tale sensore viene sviluppato come

Figura 4.5: Gerarchia dei sensori di UsarSim, versione 0.1



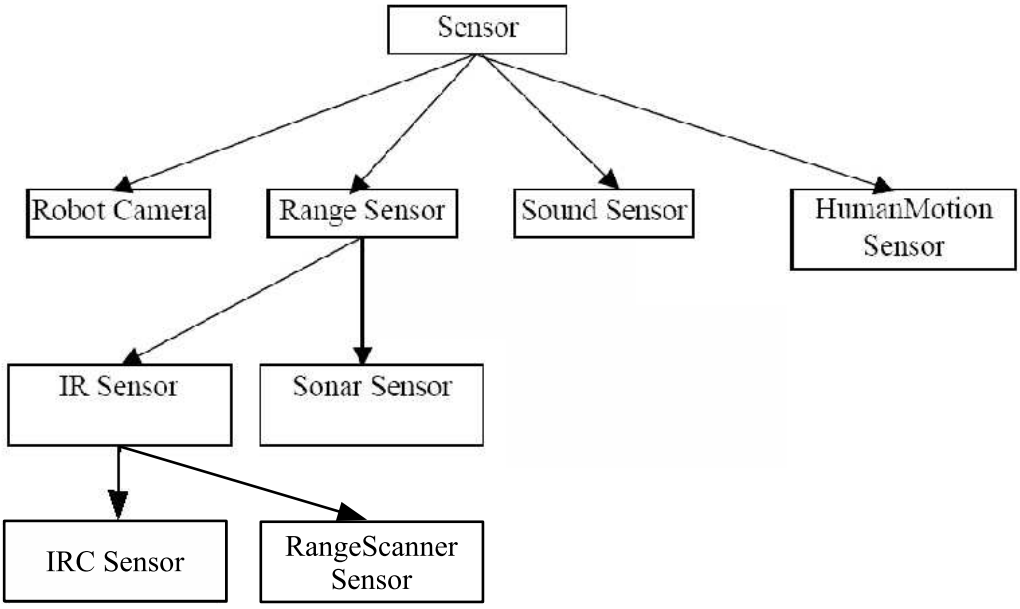
estensione del Range Sensor incorporando le limitazioni fisiche dei raggi luminosi.

In questa ottica il sensore IR è stato fornito della capacità di oltrepassare le superfici trasparenti, realizzata a livello di codice discriminando sulla proprietà *material* presente in ogni oggetto di Unreal, impostata a *trans* per gli oggetti trasparenti. Se il raggio tracciato dal sensore IR incontra un oggetto trasparente, viene propagato oltre l'oggetto, finché non incontra un oggetto opaco o viene raggiunta la distanza massima.

La soluzione è stata un ricollocamento del RangeScanner, come sotto-classe del sensore IR, minimizzando i cambiamenti apportati al codice (la sola riga d'intestazione). Il codice attuale del RangeScanner è mostrato nell'appendice A.3.

La modifica alla gerarchia dei sensori è stata proposta alla comunità che sviluppa UsarSim, accettata ed inserita nella release definita convenzionalmente 0.2, utilizzata in occasione della presentazione ufficiale di UsarSim nella manifestazione RoboCup 2005 ad Osaka. La gerarchia definitiva dei

Figura 4.6: Gerarchia dei sensori di UsarSim, versione 0.2



sensori nella release 0.2 è mostrata nella Figura 4.6.

4.4.2 Esecuzione del test

L'ambiente di test è stato scelto su misura per il TMRV. Ho predisposto un'arena simulata di tipo planare, riutilizzando alcuni componenti dell'Arena Gialla modellata in UsarSim (mostrata nella Figura 2.2). L'ambiente era composto essenzialmente di due stanze separate da un insieme di pannelli verticali di materiale trasparente, sistemati in modo tale da impedire il passaggio da una parte all'altra; un dettaglio dell'arena è mostrato in Figura 4.7.

Il test è stato eseguito utilizzando due diversi PC connessi in rete locale. Nel primo computer è stata gestita la simulazione lato server, tramite la coppia coordinata di programmi UsarSim ed *ImageServer*, mentre nel secondo PC è stata caricata la piattaforma Spqr-Rdk, configurata per l'u-

Figura 4.7: Il nostro robot alle prese con una superficie trasparente

tilizzo in simulazione, con attivi i sottosistemi di localizzazione, mapping, navigazione, esplorazione autonoma e, naturalmente, il TMRV.

Considerando la struttura dell'arena e la natura del *Sottosistema di Esplorazione Autonoma* (basato sulle frontiere), si prevedeva dal robot simulato il seguente comportamento:

- esplorazione completa del primo ambiente;
- riconoscimento della barriera di vetri;
- mappatura parziale del secondo ambiente (non accessibile);
- stop del sistema per l'esaurimento delle frontiere disponibili.

Il robot è stato posto nella stanza più ampia dell'arena, con orientamento tale da avere i pannelli trasparenti alle proprie spalle. Il test è iniziato attivando il Task di esplorazione autonoma, tramite il quale il robot ha iniziato a muoversi verso le frontiere inesplorate, individuando i confini della stanza. In seguito il robot ha compiuto un movimento ad arco verso destra, fino a trovarsi davanti ai pannelli trasparenti. A questo punto il TMRV ha individuato parte dei pannelli (i più vicini), portati in luce dalle

letture dei sonar, mentre una porzione della seconda stanza veniva mappata tramite il laser. Rimaneva una sola frontiera inesplorata, alla destra del robot, rivelatasi anch'essa ostruita da pannelli trasparenti. A questo punto, esaurite le frontiere esplorabili, il robot si è fermato.

La mappa generata da Spqr-Rdk ha riprodotto fedelmente la forma e le dimensioni della prima stanza, limitandosi a tratteggiare una porzione della seconda, per via dei limiti imposti alla navigazione dalla struttura dell'ambiente.

Per tutta la durata del test il TMRV si è comportato correttamente, individuando tutte le superfici trasparenti sfuggite allo scanner laser, con una valida approssimazione delle loro dimensioni, ed inserendole come ostacoli nella mappa dell'ambiente. Il funzionamento senza errori del TMRV ha mostrato come la simulazione dello scanner laser, dell'anello di sonar e della movimentazione del robot fosse accurata e nel complesso coerente. Il ritardo nella trasmissione dei dati, dovuto alla trasmissione tramite TCP/IP, non ha causato alcun problema di sincronizzazione e fusione dei dati provenienti dai sensori.

In conclusione UsarSim ha dimostrato di poter gestire correttamente una simulazione complessa, riproducendo realisticamente il comportamento del robot e dei sensori.

La necessità di adeguare il comportamento del sensore simulato RangeScanner in presenza di superfici trasparenti ha suscitato l'interesse nello studio di situazioni analoghe, ad esempio il comportamento in presenza di superfici totalmente riflettenti, come gli specchi. La corretta modellazione dei dati forniti dai sensori in tale contesto si tradurrà in un valido ausilio nella progettazione di un sistema di logiche decisionali che percepisca correttamente ed interpreti la natura di tali superfici.

Capitolo 5

Applicazione: Ostacoli invisibili ai sensori

In questo e nel prossimo capitolo presento due applicazioni che ho sviluppato all'interno del framework Spqr-Rdk, utilizzando UsarSim come ambiente di simulazione. La prima applicazione riguarda il problema degli ostacoli invisibili ai sensori, che se non evitati possono causare uno stallo permanente del robot. La seconda applicazione è invece relativa alla pianificazione del moto su terreno sconnesso, con l'intento di fornire al pianificatore informazioni sulla difficoltà di attraversamento di un'area e di cercare, quando possibile, un percorso sul terreno migliore.

5.1 Introduzione al problema affrontato

Nell'esperienza quotidiana nel campo dell'Esplorazione Autonoma nel contesto di soccorso robotico emerge frequentemente il problema degli stalli del robot dovuti ad ostacoli non rilevati dai sensori, dove per "stallo" si intende il blocco del robot causato dal fallimento di un'azione di movimento.

All'interno del framework Spqr-Rdk il mapping è eseguito esclusivamente sulla base delle letture del laser scanner, posto a circa 35 cm di altezza dal suolo. Di conseguenza una serie di ostacoli come tavolini bassi, travi

e tubi sul pavimento risultano invisibili a tale sensore e non possono essere mappati. Nel seguito tali oggetti saranno chiamati anche *ostacoli da impatto*, in quanto generalmente vengono rilevati tramite un urto.

In molti casi, in seguito all'impatto con un ostacolo invisibile, il robot rischia di entrare in una situazione di stallo permanentemente, cercando ripetutamente di eseguire un movimento fallace, che lo porta nuovamente ad urtare un ostacolo non previsto. A volte persino il braccio o la gamba di una vittima, se non precedentemente rilevati dalla visione, possono portare il robot in situazione di stallo.

A fronte di questa situazione si è deciso di dotare Spqr-Rdk di un *Sottosistema di Recupero dagli Stalli*, utilizzando UsarSim come ambiente di test privilegiato. L'utilizzo del simulatore è stato motivato dalla necessità di preservare l'integrità del robot reale in vista dei test del sottosistema, che avrebbero richiesto urti ripetuti contro ostacoli di vario tipo.

UsarSim si è dimostrato estremamente versatile, fornendo una serie di ostacoli fissi e mobili già nelle arene standard e rendendo semplice la creazione e l'inserimento di nuovi ostacoli.

5.2 Lavori correlati

L'approccio classico al problema degli stalli prevede l'utilizzo di sensori di sfioramento di varie tipologie e forme. In [15] viene descritta l'implementazione di un *bumper* a copertura totale, in grado di misurare la forza d'impatto, su di un robot cilindrico. In [16] viene descritto un *baffo* manovrabile allo scopo di misurare il profilo degli oggetti con cui viene a contatto, utilizzato per identificare forme note ed evitare ostacoli durante il moto.

L'estensione ed il posizionamento dei sensori di sfioramento è cruciale nel determinare la quantità e qualità delle informazioni ricavabili da ogni impatto. Anche nelle migliori ipotesi di utilizzo, l'impossibilità di coprire con appositi rilevatori ogni parte del robot soggetta ad impatto determina

una zona d'incertezza ineliminabile nella comprensione degli impatti.

Molti robot reali sono in grado di segnalare lo stallo dei motori utilizzati per la movimentazione[1]. In genere la presenza o meno di tali segnali di stallo viene confrontata con le informazioni fornite dai sensori di sfioramento per definire approssimativamente la posizione dell'oggetto impattato.

Purtroppo non tutti i robot forniscono un segnale di stallo, o non discriminano tra quale motore lo abbia generato. Queste evidenze esprimono come il tracciamento degli stalli e delle collisioni sia generalmente grossolano.

Un approccio diverso all'individuazione degli ostacoli invisibili è discusso in [17, 18], dove viene presentato un robot mobile utilizzato come guida all'interno di un museo. Il robot viene fornito di una mappa dettagliata dell'ambiente, sulla quale sono indicati sia gli ostacoli che è in grado di tracciare tramite i sensori, quali i muri e gli oggetti opachi, sia gli ostacoli che non è in grado di rilevare, come ad esempio dei pannelli di vetro.

In questo contesto il problema di evitare ostacoli invisibili, ma in posizioni note, ricade nel campo della *collision avoidance*; tale problema viene affrontato imponendo limitando la velocità del robot con un fattore proporzionale all'imprecisione della localizzazione nell'ambiente.

Purtroppo tale tecnica è inservibile nel contesto USAR, nel quale la mappa dell'ambiente non è nota a priori.

5.3 Descrizione della tecnica

La tecnica utilizzata si basa sull'unico indizio ricavabile dall'urto con un ostacolo invisibile ai sensori: una drastica riduzione della velocità del robot. Calcolata una stima della velocità attuale del robot, viene mantenuto sotto controllo lo scarto tra tale velocità e quella desiderata dal sistema

[1] Ad esempio l'informazione di stallo dei motori del robot Pioneer AT-3 è direttamente accessibile tramite i driver Aria.

Algoritmo 1 Sottosistema di Recupero dagli Stalli

Input: VelocitáDes, PosaAttuale, PosaPrec

loop
VelocitáAtt ← StimaVelocitáAttuale(PosaAttuale, PosaPrec)
CondizioniStallo ← ControlloImpatto(VelocitáDes, VelocitáAtt)
UpdateCatenaPreallarme()
DisegnoOstacoli()
ControlloTimeoutOstacoli()
end loop

di navigazione (autonomo o teleoperato). Ogni volta che lo scarto supera una determinata soglia viene attivata una catena di stati di preallarme, utilizzata per distinguere gli errori di valutazione (falsi positivi) dai reali impatti con un ostacolo invisibile.

Se il robot non riesce a raggiungere la velocità desiderata (segno che è bloccato da un ostacolo) e lo scarto permane nel tempo per una serie sufficiente di cicli, viene superato l'ultimo stato della catena di preallearme; a questo punto viene inserito un ostacolo nella mappa, in una posizione stimata dalle caratteristiche dell'impatto, e viene lanciato un avvertimento al *Sottosistema di Pianificazione del Moto*, che si occuperà di ripianificare l'esplorazione dell'ambiente tenendo conto del nuovo ostacolo.

Ogni ostacolo aggiunto in questo modo alla mappa viene dotato di un timeout al termine del quale è rimosso; questo accorgimento diminuisce la complessità dell'algoritmo in termini di memoria e tempo computazionale e soprattutto ne aumenta la tolleranza ai numerosi falsi positivi.

L'Algoritmo 1 descrive il comportamento del *Sottosistema di Recupero dagli Stalli*; in maiuscoletto sono evidenziate le fasi principali in cui può essere suddivisa la tecnica:

1. stima della velocità attuale del robot;
2. controllo della possibilità di un impatto, tramite la stima dello scarto tra la velocità attuale e quella desiderata;

3. aggiornamento della catena di preallarme dello stallo ed inserimento degli ostacoli rilevati nella *coda di disegno* e nella *coda di timeout*;
4. disegno sulla mappa degli ostacoli presenti nella *coda di disegno*;
5. rimozione dalla mappa degli ostacoli nella *coda di timeout* che abbiano oltrepassato il tempo limite.

Nel seguito verrà approfondita ogni fase, con riferimento alle funzionalità fornite e all'integrazione con gli altri sottosistemi del framework Spqr-Rdk; un paragrafo finale verrà dedicato all'implementazione della *ripianificazione veloce* all'interno del *Sottosistema di Pianificazione del Moto*.

5.3.1 Stima della velocità attuale del robot

Riguardo il primo passo, la velocità attuale del robot viene stimata sulla base delle informazioni fornite dal *Sottosistema di Localizzazione e Mapping*. Tale sottosistema utilizza un algoritmo di *scan matching*, descritto nel paragrafo 3.4.1, che permette di stimare la posizione del robot durante la costruzione della mappa.

Monitorando i dati di posizione (e i relativi timestamp) forniti dallo *scan matcher* è possibile stimare le componenti lineare ed angolare della velocità attuale del robot.

L'Algoritmo 2 approssima lo spostamento come la somma di un movimento traslatorio e di un movimento rotatorio. Tale approssimazione è ragionevolmente valida se le pose ed i tempi considerati sono sufficientemente vicini.

Nel dominio in esame lo scarto temporale tra due pose è proporzionale al tempo tra due letture dello scanner laser, intorno al decimo di secondo. Considerata una velocità massima del robot di circa 0,2 m/s, la distanza tra due pose rimane nei limiti di 2 cm, un valore sufficientemente basso da consentire l'approssimazione utilizzata.

Algoritmo 2 Funzione StimaVelocitàAttuale

Input: PosaAttuale,PosaPrec */* Pose stimate dallo scan matcher */*
Output: VelocitáAngolare, VelocitáLineare

DeltaTempo ← PosaAttuale.timestamp − PosaPrec.timestamp
if (DeltaTempo = 0) **then**
    Return */* Letture non valide */*
endif
VelocitáAngolare ← (PosaAttuale.angolo − PosaPrec.angolo) / DeltaTempo
DeltaMovimento ← distanza(PosaAttuale, PosaPrec)
VelocitáLineare ← DeltaMovimento / DeltaTempo
DeltaY ← (PosaAttuale.y − PosaPrec.y)
DeltaX ← (PosaAttuale.x − PosaPrec.x)
DeltaAngolo ← arctan(DeltaY / DeltaX)
DeltaRotazione ← DeltaAngolo − PosaPrec.angolo
if (|DeltaRotazione| > $\pi/2$) **then**
    VelocitáLineare ← −VelocitáLineare
endif

A questo punto l'analisi del movimento compiuto si riduce al calcolo delle velocità angolare e lineare. La velocità angolare si ottiene dal semplice rapporto tra lo scarto angolare delle due pose e l'arco di tempo considerato.

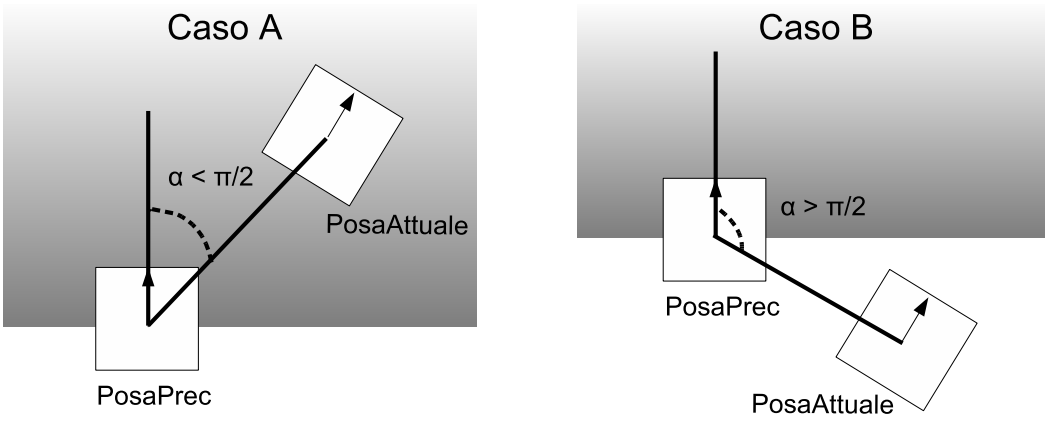
La velocità lineare è calcolata in due passi, relativi al valore assoluto ed al segno. Il valore assoluto è calcolato come rapporto tra la distanza delle due pose e l'intervallo di tempo trascorso, due misure sempre positive.

Il segno è considerato positivo se la posizione attuale del robot è più avanzata rispetto alla posizione precedente, cioè se la posizione attuale si trova nel semipiano descritto dalla linea normale all'angolo di posa precedente e dalla direzione frontale del robot.

A livello analitico, il segno viene considerato positivo se il modulo della differenza tra l'angolo formato tra l'orientazione precedente del robot e la direzione del movimento eseguiti è minore dell'angolo retto, altrimenti è considerato negativo.

Nella Figura 5.1 vengono esemplificati i due casi di velocità lineare

Figura 5.1: Calcolo del segno nella stima della velocità lineare



positiva (caso A, a sinistra) e velocità lineare negativa (caso B, a destra).

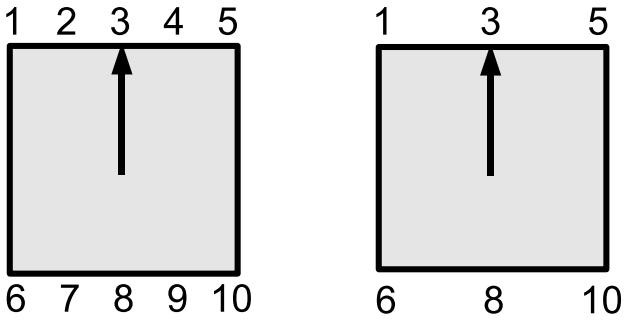
5.3.2 Controllo della possibilità di impatto con ostacoli non previsti

Calcolata una stima della velocità attuale, nelle sue componenti lineare ed angolare, l'algoritmo confronta tali valori con la velocità desiderata, che ha costituito l'origine della movimentazione del robot. La rilevazione di uno scarto significativo tra tali valori implica necessariamente una difficoltà nel movimento dovuta ad un ostacolo non previsto.

La possibilità di impatto sulla superficie del robot viene monitorata attraverso una matrice di condizioni. In virtù della non olonomia del robot (impedimento di movimenti laterali diretti), non è possibile monitorare l'impatto con oggetti posti lateralmente al robot, in quanto non è possibile stimare la velocità lineare in direzione laterale. Di conseguenza sono monitorate solamente le posizioni anteriori e posteriori al robot.

Nell'implementazione correntemente utilizzata, la matrice di condizioni è di dimensioni 2x5 e monitorizza, come mostrato in Figura 5.2 (a sinistra) cinque posizioni di impatto frontale ed altre cinque per gli impatti

Figura 5.2: Posizioni di stallo possibili, in NORMALMODE (a sinistra) e SIMPLEMODE (a destra)



posteriori; le posizioni sono state numerate sequenzialmente per consentire una facile identificazione nella trattazione seguente. Nella stessa Figura, a destra, è mostrata una modalità semplificata di monitoraggio, detta SIMPLEMODE, utilizzata in situazioni di carico pesante sul sistema, in cui non vengono sorvegliate posizioni intermedie ma solamente quelle centrali e laterali.

Ogni posizione di stallo viene controllata tramite una congiunzione di condizioni ricavate solamente dai rapporti tra le velocità (lineare ed angolare) desiderate e quelle stimate dal comportamento del robot. Tali condizioni utilizzano una serie di parametri di soglia che devono essere tarati in base all'affidabilità delle stime di velocità ricavate nella fase precedente ed alle velocità massime raggiungibili dal robot nei tre tipi di movimento lineare frontale, lineare posteriore (retromarcia) e rotatorio.

E' importante notare che il processo di taratura dei parametri non dipende dalle condizioni dell'ambiente ma solamente dall'algoritmo di localizzazione e dal tipo di robot utilizzati. Nei paragrafi seguenti sono indicati i valori di default associati all'esecuzione sul robot Pioneer 3-AT, con i dati di localizzazione forniti dal *Sottosistema di Localizzazione e Mapping* della

Tabella 5.1: Parametri di controllo della posizione frontale centrale

Parametro	**Descrizione**	**Default**
SpeedZeroTolerance	tolleranza sul valore di velocità minima riscontrato	5 mm/s
SpeedZeroMin	velocità minima desiderata	15 mm/s

piattaforma Spqr-Rdk.

Nel seguito esamino separatamente le condizioni ed i parametri utilizzati per monitorare ogni posizione di possibile impatto.

5.3.2.1 Posizioni centrali

Considerata la simmetria del robot, le equazioni che descrivono le condizioni di stallo centrale nella parte anteriore e posteriore sono identiche, a meno di cambi di segno. Considerando quindi la posizione 3, relativa allo stallo centrale frontale, il controllo viene operato secondo il sistema

$$FrontalCentralCondition = \begin{cases} DesSpeed > 0 \\ |DesSpeed| \geq SpeedZeroMin \\ |Speed| < SpeedZeroTolerance \end{cases}$$

La condizione viene verificata nel caso in cui sia desiderata una velocità lineare positiva (di valore maggiore ad una soglia minima) e sia riscontrato un valore di velocità lineare al di sotto di una soglia di tolleranza.

La Tabella 5.1 descrive i parametri utilizzati nel sistema di equazioni.

5.3.2.2 Posizioni laterali

Per la simmetria del dominio, anche le condizioni laterali (casi 1, 5, 6, 10 nella Figura 5.2) vengono monitorate utilizzando le stesse equazioni, a meno di cambi di segno. Il modello prevede l'unione di tre casi distinti.

Considerando ad esempio la posizione 1, cioè uno stallo frontale sinistro, abbiamo

$$FrontalLeftCondition = FLC1 \cup FLC2 \cup FLC3$$

Nel primo caso il moto è desiderato solamente lineare ma viene prodotta una componente angolare:

$$FLC1 = \begin{cases} DesSpeed \geq 0 \\ AttJog < -JogZeroTolerance \\ |DesJog| \leq DesJogZeroMin \end{cases}$$

Nel secondo caso è desiderato un moto solamente angolare ma viene prodotta una componente lineare:

$$FLC2 = \begin{cases} DesSpeed = 0 \\ DesJog < 0 \\ AttSpeed < -SpeedZeroLowerBound \end{cases}$$

Il terzo caso, più generale, richiede che si rilevi una velocità angolare molto maggiore rispetto alla desiderata:

$$FLC3 = \begin{cases} DesSpeed \geq 0 \\ (DesJog - AttJog) > JogTolerance \end{cases}$$

I parametri utilizzati nelle equazioni sono descritti nella Tabella 5.2 .

5.3.2.3 Posizioni intermedie

Per quanto riguarda le posizioni intermedie (numeri 2, 4, 7, 9 nella Figura 5.2, a sinistra), il loro controllo si basa esclusivamente sul'osservazione dello stato delle posizioni adiacenti. In particolare, ogni condizione intermedia è controllata a valle delle condizioni centrali e laterali, ed è verificata solamente se lo sono entrambe le condizioni adiacenti, secondo la logica per cui un ostacolo che produca un urto rilevabile contemporaneamente in posizione centrale e laterale debba essere posto in una posizione intermedia tra le due.

Tabella 5.2: Parametri di controllo della posizione frontale sinistra

Parametro	**Descrizione**	**Default**
JogZeroTolerance	tolleranza sulla velocità angolare riscontrata	2 rad/s
DesJogZeroMin	tolleranza sulla velocità angolare desiderata	0,5 rad/s
SpeedZeroLowerBound	valore minimo di velocità lineare riscontrata	25 mm/s
JogTolerance	tolleranza sullo scarto tra le velocità angolari attuale e desiderata	2 rad/s

Questo accorgimento viene utilizzato per discriminare in modo più accurato la posizione dell'ostacolo, considerando solamente gli effetti riscontrati sulle discrepanze tra le velocità desiderate e quelle attuali.

Ad esempio, se vengono verificate contemporaneamente le condizioni in posizione 1 e 3, si considera che l'oggetto impattato sia in realtà in posizione 2. Di conseguenza, la procedura di controllo delle posizioni intermedie, sovvertendo le rilevazioni precedenti, rende attiva la condizione in 2 ed inibisce le condizioni 1 e 3.

Il controllo delle posizioni intermedie è opzionale e viene disabilitato automaticamente dall'algoritmo in caso di carico pesante, ad esempio quando viene rilevato un tempo di ciclo elevato[2]; il *Sottosistema* entra allora in SimpleMode (in contrasto col NormalMode, con le posizioni intermedie attive), risparmiando risorse di calcolo in cambio di una localizzazione meno accurata degli ostacoli.

Nella Figura 5.2 a pagina 91 vengono confrontate le posizioni control-

[2] All'interno di Spqr-Rdk, il controllo delle condizioni intermedie può essere effettuato anche interattivamente dall'operatore selezionando la proprietà SimpleMode tra le opzioni del *Sottosistema di Recupero dagli Stalli*.

late rispettivamente nel NormalMode, a sinistra, e nel SimpleMode, a destra.

5.3.3 Aggiornamento della catena di preallarme

La catena di preallarme dello stallo viene utilizzata allo scopo fondamentale di scartare i falsi positivi, cioè le situazioni per cui l'algoritmo crede erroneamente di aver rilevato un ostacolo in realtà non presente.

I falsi positivi nel dominio considerato sono molto frequenti, in quanto il *Sottosistema di Recupero dagli Stalli* utilizza le stime fornite dal *Sottosistema di Localizzazione e Mapping*, i cui anche minimi errori di valutazione vengono notevolmente amplificati.

Le condizioni usuali che portano ad un falso positivo includono problemi nei movimenti, presenza di terreno dissestato (ma praticabile) ed infine le occasionali oscillazioni della posizione del robot imputabili ad errori di localizzazione.

Il modello di aggiornamento della catena di preallarme, descritto nell'Algoritmo 3, può essere descritto come un'*integrazione nel tempo* delle condizioni di stallo determinate nella fase precedente.

Ogni volta che viene verificata la condizione di impatto in una delle posizioni controllate, viene attivata la catena di preallarme relativa a quella posizione.

Nel caso in cui il robot riesca a raggiungere la velocità desiderata, le condizioni di stallo diventano invalide ed il preallarme ridiscende spontaneamente lungo la catena, fino ad estinguersi.

Altrimenti, finché il robot non riesce a raggiungere la velocità desiderata (segnale di stallo con un ostacolo imprevisto), le condizioni rimangono attive nel tempo. Se tale stato permane per una serie sufficiente di cicli, l'allarme sale lungo la catena fino a superare l'ultimo stadio. A questo punto l'algoritmo resetta la catena di preallarme ed inserisce un oggetto nella

Algoritmo 3 Funzione UpdateCatenaPreallarme

Input: PosizioniMonitorate, CondizioniStallo, StalloMotore, PosaAttuale

```
for all Pos in PosizioniMonitorate do
    if StalloMotore[Pos] then
        CatenaPreallarme[Pos] ← CatenaPreallarme[Pos] + 3
    end if
    if (CondizioniStallo[Pos]) then
        CatenaPreallarme[Pos] ← CatenaPreallarme[Pos] + 1
    else
        if (CatenaPreallarme[Pos] > 0) then
            CatenaPreallarme[Pos] ← CatenaPreallarme[Pos] − 1
        end if
        Return /* Decadimento spontaneo lungo la catena */
    end if
    if (CatenaPreallarme[Pos] > MaxPreallarme) then
        CodaDisegno.add(Pos, PosaAttuale)
        CodaTimeout.add(Pos, PosaAttuale)
        CatenaPreallarme[Pos] ← 0
    end if
end for
```

coda di disegno, che verrà utilizzata nella fase successiva per collocare sulla mappa del robot gli ostacoli incontrati.

Gli oggetti inseriti nella *coda di disegno* sono corredati di una serie di informazioni (la posizione di impatto sul robot, la posa del robot al momento dell'impatto ed il relativo timestamp), utili al posizionamento dell'ostacolo ed alla gestione del suo ciclo di vita.

La lunghezza delle catene di preallarme è definita nel parametro MAX-PREALLARME, il cui valore di default è impostato a 6 stadi.

In questa fase viene anche inserita l'elaborazione dei segnali di stallo dei motori del robot, se presenti. Considerata la maggiore attendibilità dell'informazione di stallo fornita dai motori, in caso di intercettazione di tale segnale vengono saliti più stadi nella catena di preallarme.

La gestione di tali segnali è strettamente dipendente dal grado di dettaglio che il robot è in grado di fornire sui motori: maggiore è il dettaglio, più semplice è stabilire la posizione di impatto sul robot.

In mancanza di informazioni dettagliate, come nel caso di un solo bit di stallo generico, l'algoritmo non è in grado di localizzare la posizione di impatto; di conseguenza viene generato un debole allarme in tutte le posizioni, aumentando di un solo stadio lo stato di tutte le catene di preallarme.

5.3.4 Disegno sulla mappa dell'ostacolo impattato

Come si è visto nella descrizione delle fasi precedenti, gli allarmi di stallo che riescono a superare la catena di preallarme vengono inseriti nella *coda di disegno*, con tutte le informazioni necessarie a trattarli.

Ovviamente non è possibile disegnare sulla mappa dal *Sottosistema di Recupero dagli Stalli* l'intero profilo dell'ostacolo impattato, per l'esiguità delle informazioni disponibili (un'approssimazione della forma di un oggetto esteso potrà essere ricavata solamente tramite impatti ripetuti). Ciò che è disegnato sulla mappa è un semplice segmento di larghezza arbitraria

(il cui valore di default è un terzo della larghezza del robot), centrato sul probabile punto d'impatto tra il robot e l'ostacolo.

Tra le alternative di disegno disponibili, per forma e dimensione, il segmento è stato scelto per la sua semplicità e per la capacità di rappresentare in maniera non ridondante tutta l'informazione ricavata dall'impatto.

La fase di disegno prevede l'estrazione di tutti gli oggetti presenti nella *coda di disegno* , se non vuota, e compie una serie di operazioni su ciascuno fino ad ottenere la sua rappresentazione sulla mappa pubblica dell'Spqr-Rdk.

Il processamento di ogni oggetto prevede i seguenti passi:

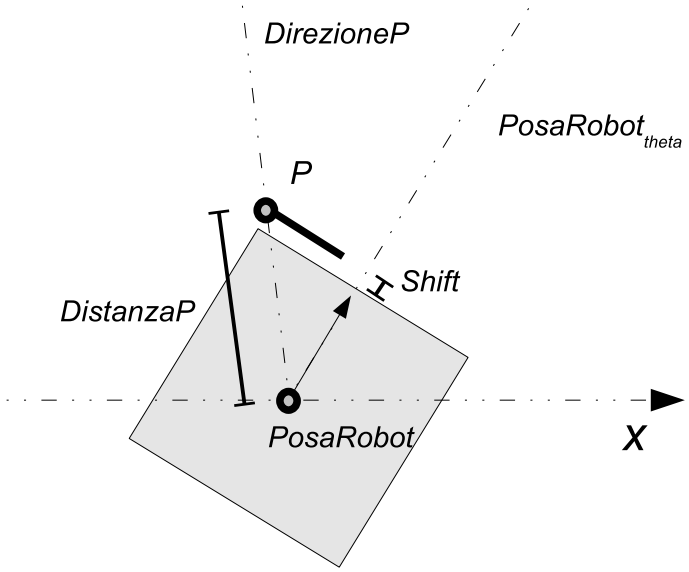
1. il *calcolo della posizione* più probabile dell'oggetto nell'ambiente;
2. la *trasformazione di coordinate* per gli estremi del segmento, da quelle dell'ambiente a quelle sulla mappa pubblica;
3. il *calcolo dei pixel* della mappa pubblica sui quali verrà disegnato il segmento;
4. il *controllo* della possibilità di modifica per ogni pixel individuato;
5. l'*invio* dei pixel modificati alla mappa pubblica di Spqr-Rdk.

Il *calcolo della posizione*, al punto 1, è eseguito solamente per i due estremi del segmento che rappresenta l'ostacolo. La procedura di calcolo è basata sulla trigonometria ed utilizza come variabili il punto di impatto e la posa del robot al momento della rilevazione.

Come esempio è mostrato il calcolo della posizione P(x,y) dell'estremo sinistro del segmento di un ostacolo in posizione frontale sinistra (la Figura 5.3 mostra le variabili in gioco):

$$DirezioneP = \frac{\pi * RobotWidth}{4 * (RobotHeight + Shift)}$$

Figura 5.3: Calcolo della posizione dell'estremo di un ostacolo in posizione frontale sinistra



$$DistanzaP = \frac{RobotHeight + Shift}{2 * \cos(DirezioneP)}$$

$$P_x = PosaRobot_x + DistanzaP * \cos(PosaRobot_{theta} + DirezioneP)$$
$$P_y = PosaRobot_y + DistanzaP * \sin(PosaRobot_{theta} + DirezioneP)$$

Il parametro SHIFT viene utilizzato per impostare la distanza tra l'ostacolo ed il robot e può essere ridefinito dinamicamente durante l'esecuzione dell'algoritmo.

La procedura di *controllo* della possibilità di modifica di un pixel è progettata allo scopo di evitare il caso in cui un ostacolo da impatto, la cui presenza è limitata nel tempo, possa sovrascrivere un ostacolo permanente.

Tale controllo, in linea generale, è eseguito ogni volta che un Task inserisce informazioni nella mappa pubblica, e viene operato in base al colore:

Tabella 5.3: Significato e priorità dei colori utilizzati nella mappa pubblica dell'Spqr-Rdk

Colore	**Priorità**	**Significato**
Nero	I	indica gli ostacoli permanenti, rilevati dallo scanner laser
Viola	II	indica la presenza di ostacoli da impatto
Verde	III	indica un'informazione di terreno sconnesso, inserita dall'algoritmo descritto nel capitolo 6
Bianco	-	indica lo spazio libero
Blu	-	indica le zone della mappa sconosciute al robot

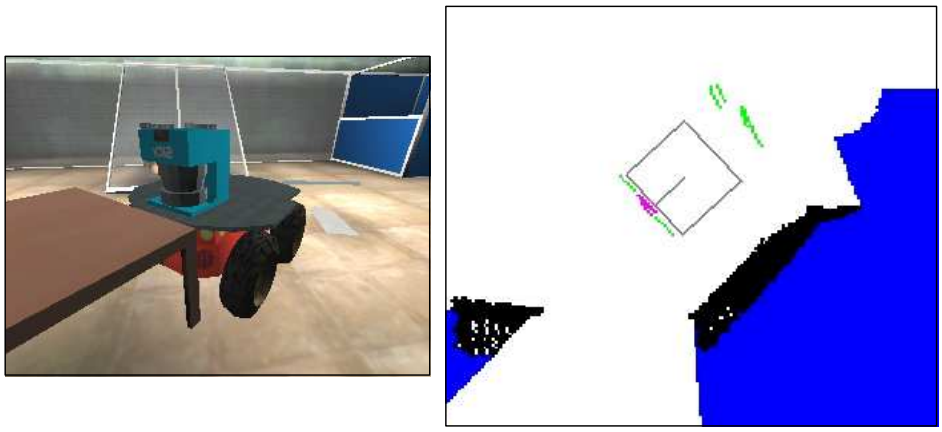
considerata la scala di priorità dei colori (mostrata nella Tabella 5.3) definita nell'Spqr-Rdk, viene controllato che la priorità assegnata al colore attuale del pixel sulla mappa sia maggiore della priorità del colore da inserire.

Nel caso in esame, tale procedura si riduce al controllo che il colore attuale sia diverso dal nero, cioè che la segnalazione di un ostacolo da impatto, la cui presenza è limitata nel tempo, non vada a sovrascrivere un ostacolo permanente.

Nella Figura 5.4, a sinistra, è mostrato l'urto tra il robot ed un ostacolo al di sotto del piano di lettura dello scanner laser; a destra è mostrata la mappa pubblica costruita nella piattaforma Spqr-Rdk. Il significato dei colori utilizzati nella mappa è descritto nella Tabella 5.3, accanto all'informazione di priorità del colore.

L'ultima operazione compiuta nella fase di disegno, se è stato processato almeno un ostacolo, consiste nella generazione di un avvertimento al *Sottosistema di Pianificazione del Moto* per l'avvio di una ripianificazione veloce. L'avvertimento comprende un informazione booleana (PosizioneFrontale) per distinguere gli impatti frontali da quelli posteriori, che il pianificatore tratta in maniera opposta.

Figura 5.4: Esempio di esecuzione del *Sottosistema di Recupero dagli Stalli*



La scelta di distinguere sulla mappa pubblica gli ostacoli permanenti dagli ostacoli da impatto, anche visivamente, con diversi colori, è stata motivata dal voler fornire ai Task dell'Spqr-Rdk tutte le informazioni disponibili sullo stato del sistema.

La conseguenza diretta di tale scelta è stata la necessità di implementare negli altri Task il trattamento esplicito della nuova tipologia di ostacolo.

A tal proposito ho modificato, ad esempio, il *Sottosistema di Pianificazione del Moto*, estendendo la funzione di *controllo delle collisioni* nell'algoritmo di generazione del percorso tramite la verifica della collisione anche con gli ostacoli invisibili ai sensori.

5.3.5 Controllo dei timeout degli ostacoli aggiunti

Allo scopo di tenere traccia di tutti gli ostacoli da impatto inseriti nella mappa pubblica, al momento dell'inserimento viene creato un nuovo oggetto nella *coda di timeout*.

L'ultima fase del ciclo implementato nel *Sottosistema di Recupero dagli Stalli* riguarda il controllo della *coda di timeout* alla ricerca degli ostacoli per cui sia scaduto il tempo di vita.

Algoritmo 4 Funzione RipianificazioneVeloce
Input: PianoCorrente, AvvertimentoAlPianificatore, PosizioneFrontale

```
if (AvvertimentoAlPianificatore) then
    PianoCorrente ← Reset
    if (PosizioneOstacolo = PosizioneFrontale) then
        VelocitáDesiderata ← (−VelocitáMassima)
    else
        VelocitáDesiderata ← VelocitáMassima
    end if
    AvvertimentoAlPianificatore ← false
end if
```

Per ogni timeout rilevato, l'algoritmo ricostruisce i pixel sulla mappa appartenenti al segmento in esame e si occupa di rimuoverli dalla mappa pubblica, sostituendo al loro posto spazio libero.

Analogamente alla procedura di disegno, quella di eliminazione è vincolata ad un controllo preventivo della possibilità di modifica del pixel: viene abilitata solamente la cancellazione di pixel di colore viola, corrispondenti agli ostacoli permanenti.

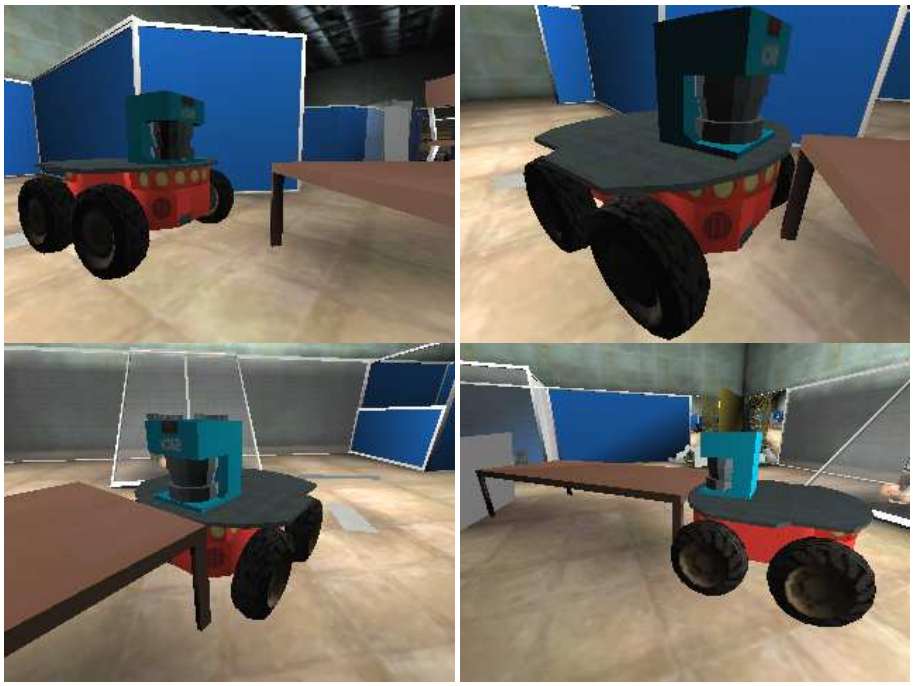
5.3.6 Ripianificazione veloce nel *Sottosistema di Pianificazione del Moto*

L'esecuzione del *Sottosistema di Recupero dagli Stalli* inserisce nuovi ostacoli sulla mappa interna del robot. Nel contesto dell'esplorazione autonoma dell'ambiente la generazione di tali ostacoli invalida il piano corrente, rendendo necessaria una ripianificazione a posteriori, con conseguente spreco di una serie di cicli di elaborazione.

Nell'intento di rendere il robot più responsivo in caso di stallo ho deciso di modificare il *Sottosistema di Pianificazione del Moto* (presentato in 3.4.3) in modo da intercettare l'inserimento di un ostacolo invisibile sulla mappa e rispondere adeguatamente.

L'Algoritmo 4 mostra il comportamento adottato: tramite il controllo

Figura 5.5: Il nostro robot alle prese con un piccolo tavolo



di due variabili booleane (impostate dal SRS) viene operato il reset del piano corrente ed un movimento per allontanarsi dall'ostacolo, allo scopo allargare la visuale del robot e rendere più agevole la costruzione del nuovo piano.

Nel ciclo successivo, in virtù del reset, verrà eseguita una nuova pianificazione a partire dalla posizione raggiunta verso l'obiettivo finale, che terrà conto dell'ostacolo inserito.

5.4 Test dell'applicazione

Nei test il robot è stato fatto interagire con una serie di oggetti, tra cui un piccolo tavolo (mostrato in Figura 5.5), un tubo ed una piccola rampa.

Nella versione attuale di UsarSim non è implementata la generazione

dei segnali di stallo dei motori, quindi per il test di tale *feature* è stato necessario attendere le prove sul robot reale.

Nei test preliminari sono stati configurati i valori più appropriati per i parametri necessari al *Sottosistema di Recupero degli Stalli* utilizzando un sistema di movimentazione manuale del robot. Nei test successivi, la movimentazione del robot è stata resa completamente autonoma, grazie all'utilizzo del *Sottosistema di Pianificazione del Moto* integrato nella piattaforma Spqr-Rdk.

L'esecuzione simulata dell'impatto del robot con oggetti di varia forma e dimensione, a velocità diverse e in posizioni diverse ha permesso di apprezzare la robustezza e l'adattabilità dell'algoritmo, che ha reagito positivamente nella maggioranza dei casi, identificando correttamente gli ostacoli e permettendone l'aggiramento da parte del *Sottosistema di Pianificazione del Moto*.

In alcune situazioni l'ostacolo non è stato riconosciuto al primo impatto; in ogni caso, l'algoritmo è sempre riuscito nell'identificazione in un secondo momento, quando il pianificatore del moto lo ha portato ad urtare nuovamente l'ostacolo.

Nell'esecuzione dei test l'algoritmo si è dimostrato efficace ed efficiente. Il tempo di elaborazione del *Sottosistema di Recupero dagli Stalli* si è rivelato pressochè costante.

In seguito il *Sottosistema di Recupero dagli Stalli* è stato testato sul robot reale. In questo contesto è stata testata l'integrazione con la *feature* di intercettazione dei segnali di stallo provenienti dai motori. Tale *feature* ha migliorato la responsività dell'algoritmo, riducendo di circa un terzo il tempo necessario a riconoscere uno stallo nei casi in cui era effettivamente inviato un segnale di stallo dei motori.

E' comunque da rilevare che in buona parte dei test effettuati l'impatto con un ostacolo non comportava il blocco delle ruote, che continuavano a slittare sul terreno senza far scattare lo stallo dei motori.

Altre volte il robot ha inviato il segnale di stallo arbitrariamente, du-

rante rotazioni del robot a bassa velocità angolare, movimento che sforza i motori su terreno non perfettamente liscio, in totale assenza di ostacoli nell'ambiente circostante.

Tali informazioni sperimentali confermano l'intuizione che l'informazione di stallo, per quanto utile, sia un fattore da mantenere assolutamente in secondo piano nell'analisi degli stalli del robot.

5.5 Sintesi del lavoro svolto

In conclusione, lo sviluppo del *Sottosistema di Recupero dagli Stalli* ha permesso di affrontare e risolvere un problema spesso sottovalutato nello sviluppo di un sistema di controllo robotico, quello di rendere la movimentazione del robot robusta rispetto ai limiti dei sistemi di rilevamento adottati.

Attualmente il *Sottosistema di Recupero dagli Stalli* è inserito nella configurazione standard del robot utilizzato dal dipartimento nelle operazioni simulate e nelle competizioni internazionali dedicate al soccorso.

Un miglioramento possibile dell'algoritmo di rilevamento degli stalli consiste nel ridurre il numero di parametri da cui dipende e renderli il più possibile adattivi.

Per quanto riguarda il simulatore UsarSim, è necessario modellare lo stallo dei motori del robot, aggiungendo l'emissione di un segnale apposito ogni volta in cui tale situazione si verifichi.

Capitolo 6

Applicazione: Pianificare il moto su terreno sconnesso

In questo capitolo viene descritta un'applicazione riguardante la pianificazione del moto su terreno sconnesso, che ho sviluppato all'interno della piattaforma Spqr-Rdk utilizzando il simulatore UsarSim. In questa applicazione il pianificatore viene integrato con informazioni sulla difficoltà di attraversamento di un ambiente e viene dotato della capacità di cercare quando possibile il percorso sul terreno migliore.

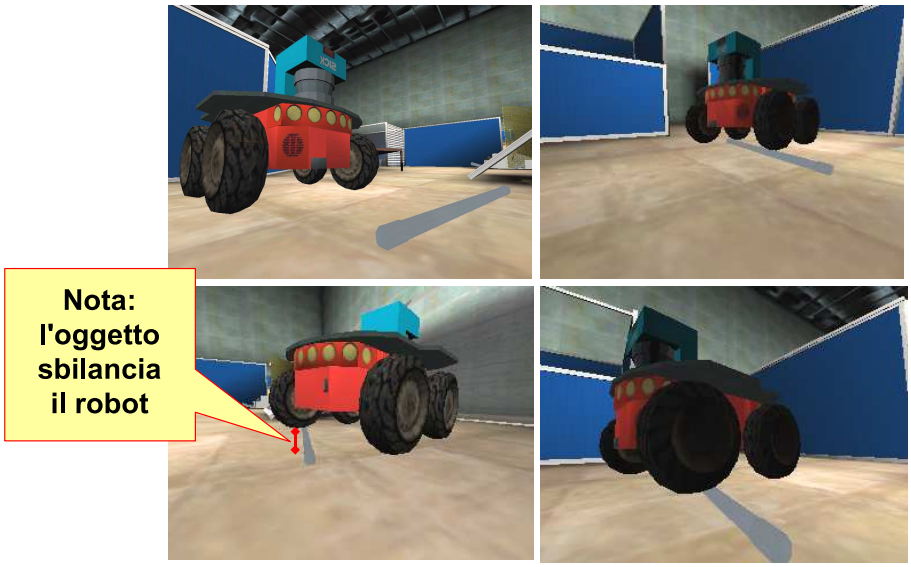
6.1 Introduzione al problema affrontato

Nello sviluppo di un sistema autonomo di navigazione robotica si postula spesso una generale omogeneità nel terreno da attraversare, assumendo che i comandi impartiti al moto del robot non debbano dipendere dalle caratteristiche del terreno e che la loro esecuzione non sia problematica.

Come si è visto anche nel capitolo precedente, nel mondo reale tali assunzioni non sono sempre valide; al contrario, è bene sviluppare un buon livello di tolleranza ai problemi di movimentazione.

Evitare il terreno sconnesso comporta una serie di benefici al robot, principalmente riguardo la stabilità delle letture dei sensori. Gli errori di

Figura 6.1: Il nostro robot alle prese con un tubo



valutazione, dovuti alla perdita di equilibrio, hanno spesso forti ripercussioni sul comportamento del robot, specialmente per quanto riguarda i sensori che utilizzano l'*assunzione 2D* (vedi il paragrafo 1.4), quali lo scanner laser o i sonar, che necessitano di un piano di navigazione stabile.

Per capire l'entità delle ripercussioni dovute all'attraversamento di un terreno accidentato mostro un esempio tratto da uno dei test dell'applicazione. Nella Figura 6.1 possiamo notare il comportamento del robot, in navigazione autonoma, nell'attraversamento di un piccolo tubo posato in terra.

Nel momento in cui il robot sale sul tubo con le ruote anteriori, viene sbilanciato verticalmente verso l'alto, con angolo di tilt α; di conseguenza il piano di lettura dello scanner laser diventa obliquo e le letture del sensore vengono generalmente aumentate di un valore proporzionale a $\cos\alpha$.

Ciò conduce ad un errore di localizzazione in quanto l'aumento della distanza dagli oggetti circostanti viene interpretato come un allontanamento.

Tale errore può in seguito ripercuotersi sul processo di mapping ed essere inoltre alimentato dagli ulteriori sbilanciamenti del sistema di riferimento del robot.

Un caso diverso riguarda gli oggetti che causano una rotazione del robot in senso orizzontale. Ad esempio si può considerare un insieme di fogli di giornale in terra. Il comportamento tipico di un robot su ruote, in questa situazione, prevede lo slittamento delle ruote che entrano in contatto coi fogli, con conseguente rotazione orizzontale del robot.

Per ovviare a tali inconvenienti risulta in generale difficile intervenire in anticipo. Una tecnica in questo senso prevede l'utilizzo di unità inerziali in grado di stabilire con precisione gli angoli di tilt e roll del robot, monitorando così ogni sbilanciamento verticale. Tali unità sono comunque poco sensibili all'angolo di pan, risultando poco utili nel monitorare gli sbilanciamenti orizzontali.

In generale, in mancanza di una soluzione omnicomprensiva al problema, è auspicabile monitorare l'esplorazione dell'ambiente, tenendo traccia delle asperità incontrate, allo scopo di evitarle nella navigazione successiva.

Sulla scorta di tale ragionamento ho sviluppato nella piattaforma Spqr-Rdk un'estensione del *Sottosistema di Pianificazione del Moto*, in grado di costruire i percorsi di navigazione privilegiando il terreno meno accidentato, in base alle informazioni acquisite durante l'esplorazione dell'ambiente.

Riguardo al contesto di test dell'applicazione, la scelta è caduta naturalmente sul simulatore UsarSim, grazie al quale è stato semplice fare interagire il robot virtuale in molteplici ambienti e situazioni modellate ad hoc, accelerando lo sviluppo e riducendo l'usura del robot reale, sul quale sono stati eseguiti solamente i test finali.

6.2 Descrizione della tecnica

La tecnica utilizzata per approcciare e risolvere il problema del moto su terreno sconnesso si basa sul tracciamento delle difficoltà nel moto.

A tal proposito ho realizzato un'estensione del *Sottosistema di Recupero dagli Stalli* di Spqr-Rdk, che tenga traccia non solo degli ostacoli da impatto, ma di tutte le situazioni in cui il moto pianificato dal robot risulti compromesso, anche in maniera lieve, dalle asperità del terreno attraversato.

Tutte le situazioni anomale vengono annotate sulla mappa pubblica tramite l'inserimento di segmenti nelle posizioni di rilevamento, in maniera simile al tracciamento degli ostacoli da impatto.

La rilevazione di tali annotazioni è operata tramite un'estensione del *Sottosistema di Pianificazione del Moto*, che inserisce tra i criteri decisionali del pianificatore un nuovo fattore relativo alla qualità del terreno da attraversare. In questo modo, nella generazione del cammino migliore verso la destinazione, il robot tenderà ad evitare le zone in cui in precedenza ha incontrato difficoltà nel moto.

6.2.1 Estensione del *Sottosistema di Recupero dagli Stalli*

La funzionalità di tracciamento delle asperità incontrate nella navigazione è stata inserita nel *Sottosistema di Recupero dagli Stalli* (abbreviato nel seguito con la sigla SRS), presentato nel capitolo 5.

Considerando la buona resa dell'algoritmo di tracciamento posto alla base del SRS, mi sono limitato a replicarne le funzionalità su una struttura dati separata (composta da una matrice di condizioni ed una matrice di stati di preallarme), adibita al controllo degli impatti con le asperità del terreno.

Allo scopo di aumentare la sensibilità sulle differenze di velocità, ho accorciato le catene di preallarme ed ho utilizzato un differente sistema di equazioni per la verifica delle condizioni d'impatto.

Tabella 6.1: Parametri nell'estensione al *Sottosistema di Recupero dagli Stalli*

Parametro	**Descrizione**	**Default**
RoughSpeedTolerance	tolleranza sul massimo del rapporto tra le velocità lineari	1,5
RoughJogTolerance	tolleranza sul massimo del rapporto tra le velocità angolari; il valore di default è maggiore rispetto al caso precedente in quanto sperimentalmente la stima della velocità angolare è più approssimativa	2
PathQualityTolerance	tolleranza sul rapporto tra segmenti disegnati e metri percorsi dal robot	0,2

Le nuove equazioni monitorano semplicemente i rapporti tra le velocità desiderate ed attuali; le condizioni vengono verificate al superamento di determinate soglie. Come esempio mostro le condizioni relative alla rilevazione di un'asperità di fronte al robot, in posizione centrale e sinistra (posizioni 3 e 1 nella Figura 5.2).

$$FrontalCentralCondition = \begin{cases} DesSpeed > 0 \\ \frac{DesSpeed}{ActSpeed} \geq RoughSpeedTolerance \end{cases}$$

$$FrontalLeftCondition = \begin{cases} DesSpeed > 0 \\ \frac{DesJog}{ActJog} \geq RoughJogTolerance \end{cases}$$

Le soglie sono impostate attraverso i due nuovi parametri RoughSpeedTolerance e RoughJogTolerance, descritti nella Tabella 6.1 .

Riguardo alla fase di disegno, ho scelto di riutilizzare la *coda di disegno* già implementata, inserendo un campo nei record delle code che discrimini gli ostacoli da impatto dalle asperità nel terreno.

Il *controllo* della possibilità di modifica dei pixel, al momento dell'inserimento sulla mappa pubblica, è operato in modo da evitare la sovrascrittura sia degli ostacoli reali che di quelli da impatto. Tale controllo rispetta la priorità tra i colori definita nella Tabella 5.3.

Nella fase di disegno ho inoltre inserito due contatori, che memorizzano rispettivamente il numero totale di segmenti disegnati ed il numero metri percorsi dal robot dall'avvio dell'applicazione[1]. Il rapporto tra queste due variabili può essere considerato un indicatore approssimativo della qualità del terreno esplorato.

Durante l'esecuzione del SRS, ogni volta che il valore di tale indicatore supera il parametro di soglia PathQualityTolerance (descritto nella Tabella 6.1), viene attivata la variabile booleana TerrenoSconnesso, in grado di abilitare l'utilizzo dell'estensione al *Sottosistema di Pianificazione del Moto*, descritta nel paragrafo successivo.

Per quanto riguarda, infine, il controllo dei timeout, la necessità di riutilizzare nel tempo le informazioni memorizzate mi ha imposto di disattivare il controllo dei timeout relativamente alle asperità del terreno.

Nelle Figure 5.5 e 6.3 (alle pagine, rispettivamente, 103 e 117) sono mostrati esempi di test sul simulatore UsarSim, in cui le difficoltà incontrate durante la navigazione vengono tracciate dal SRS tramite l'inserimento di segmenti di colore verde nella mappa pubblica del robot.

[1] L'informazione sul cammino percorso è ottenuta dal PollicinoTask dell'Spqr-Rdk.

6.2.2 Estensione del *Sottosistema di Pianificazione del Moto*

Il *Sottosistema di Pianificazione del Moto*, presentato in 3.4.3, è stato esteso per utilizzare le informazioni sulle asperità del terreno inserite dal SRS nella mappa pubblica.

In particolare è stato esteso il *Motion Planner*, cioè il Task che si occupa della pianificazione dettagliata di una traiettoria di moto in una porzione ristretta della mappa, tramite la costruzione incrementale di un *albero delle pose* (come spiegato in 3.4.3.2).

Un punto cruciale dell'algoritmo è rappresentato dal criterio di selezione utilizzato ad ogni passo per aggiungere un nodo all'albero. La procedura standard prevede il calcolo delle distanze tra il nodo e la destinazione, per ognuno dei nodi candidati, e la scelta del nodo per cui tale distanza è minima.

Il criterio di decisione esteso è mostrato nell'Algoritmo 5. Il passo preliminare prevede il calcolo del nodo più vicino alla destinazione e la scelta provvisoria di tale nodo come successore nell'albero.

Tale decisione può essere sovvertita se viene verificata una serie di condizioni in cascata. La prima condizione è che sia attiva (al valore logico `vero`) nel SRS la variabile booleana TerrenoSconnesso (descritta nel paragrafo precedente), che segnala un alto grado di dissesto del terreno finora incontrato.

Se ciò è verificato, viene calcolata per ogni nodo un'informazione di asperità, utilizzando le informazioni inserite dal SRS nella mappa pubblica. In particolare, considerate le informazioni di posizione ed orientamento indicate nel nodo e le informazioni sulla dimensione e forma del robot, viene calcolato l'insieme dei pixel coperti dal perimetro del robot in tale posa.

Tra quest'insieme di pixel viene contata la percentuale di asperità presenti; tecnicamente, considerato che l'informazione di asperità viene indi-

Algoritmo 5 Selezione del nodo successore

Input: TerrenoSconnesso, NodiCandidati

```
NodoPiùVicino ← CalcoloNodoPiùVicino()
ProssimoNodo ← NodoPiùVicino
if (TerrenoSconnesso = true) then
    AsperitàMassima ← 0
    AsperitàMinima ← ∞
    NodoPiùPulito ← None
    for all Nodo in NodiCandidati do
        AsperitàNodo ← CalcoloAsperitàNodo()
        if (AsperitàNodo > AsperitàMassima) then
            AsperitàMassima ← AsperitàNodo
        end if
        if (AsperitàNodo < AsperitàMinima) then
            NodoPiùPulito ← Nodo
        end if
    end for
    if (NodoPiùVicino ≠ NodoPiùPulito) then
        if (AsperitàMassima > RoughMinTolerance) then
            /* Scelgo un numero casuale tra 1 e 100 */
            Casualità ← NumeroCasuale(100)
            if (Casualità ≤ RandomTolerance) then
                ProssimoNodo ← NodoPiùPulito
            end if
        end if
    end if
end if
```

Tabella 6.2: Parametri utilizzati nell'Algoritmo di Selezione del nodo successore

Parametro	**Descrizione**	**Default**
RoughMinTolerance	percentuale necessaria di asperità nel nodo più "sporco"	10%
RandomTolerance	percentuale di volte per cui verrà scelto il nodo più "pulito" in favore del nodo più vicino	70%

cata sulla mappa pubblica tramite il colore, viene contata la percentuale di pixel di colore verde presenti nel perimetro del poligono[2] che indica la posa assunta dal robot in quel nodo.

Durante il calcolo delle asperità dei nodi vengono memorizzati il nodo più "pulito", cioè che presenta asperità minima, ed il valore della massima percentuale di asperità incontrata.

A questo punto il criterio di scelta tra il nodo più "pulito" ed il nodo più "vicino" (posto che tali indicatori non individuino lo stesso nodo) si avvale di due parametri, descritti nella Tabella 6.2. Viene controllato che l'asperità massima tra i nodi candidati sia superiore alla soglia RoughMinTolerance; in caso contrario, i nodi candidati sono considerati equivalenti dal punto di vista della difficoltà del terreno, e viene preferito il criterio di distanza dalla destinazione.

Se invece il controllo è superato, viene operata la scelta finale tra i due nodi. La decisione è casuale, con probabilità maggiore assegnata al nodo più "pulito" e definita nel parametro RandomTolerance.

L'utilizzo della randomicità nel passo finale dell'algoritmo è giustificato dalla necessità di evitare i minimi locali, cioè le aree "pulite" ma isolate dalla destinazione, nelle quali il robot potrebbe rimanere intrappolato senza

[2]Nell'Spqr-Rdk il robot è solitamente rappresentato tramite un rettangolo od un cerchio.

possibilità di uscita.

Un esempio semplificato di utilizzo dell'estensione del *Sottosistema di Pianificazione del Moto* è mostrato nella Figura 6.2, in cui il robot ha lo scopo di raggiungere la destinazione disegnata in rosso.

Nel primo passo, fra i tre candidati possibili viene provvisoriamente selezionato il nodo C1, più vicino alla destinazione. Immaginando attiva la variabile TerrenoSconnesso, viene attivato il calcolo del nodo che presenta il minimo di asperità, il nodo C2. Considerando superata la soglia RoughMinTolerance (in virtù dell'alto valore di asperità riscontrato per i nodi C1 e C3) e favorevole la casualità calcolata, il nodo C2 viene selezionato quale successore ed inserito nell'albero. In questo modo viene evitato il terreno sconnesso alla sinistra del robot.

Al passo successivo, da tale posizione vengono generati i nuovi nodi candidati. In questo caso viene individuato il nodo C1 sia dal criterio di distanza dalla destinazione, sia dal criterio di difficoltà di attraversamento; di conseguenza l'algoritmo seleziona direttamente tale nodo come successore e salta al passo successivo.

6.3 Test dell'applicazione

L'applicazione è stata testata in UsarSim, utilizzando le arene standard Gialla, Arancione e Rossa, descritte nel paragrafo 2.3.1 a pagina 37.

L'esecuzione dei test ha previsto l'esplorazione autonoma degli ambienti, confrontando il comportamento del robot nei casi di utilizzo o meno del software realizzato.

Inizialmente ho condotto alcuni test preliminari allo scopo di impostare i parametri utilizzati dall'applicazione; i valori correntemente utilizzati sono riportati nelle tabelle 6.1 e 6.2.

Figura 6.2: Esempio di scelta del nodo successore

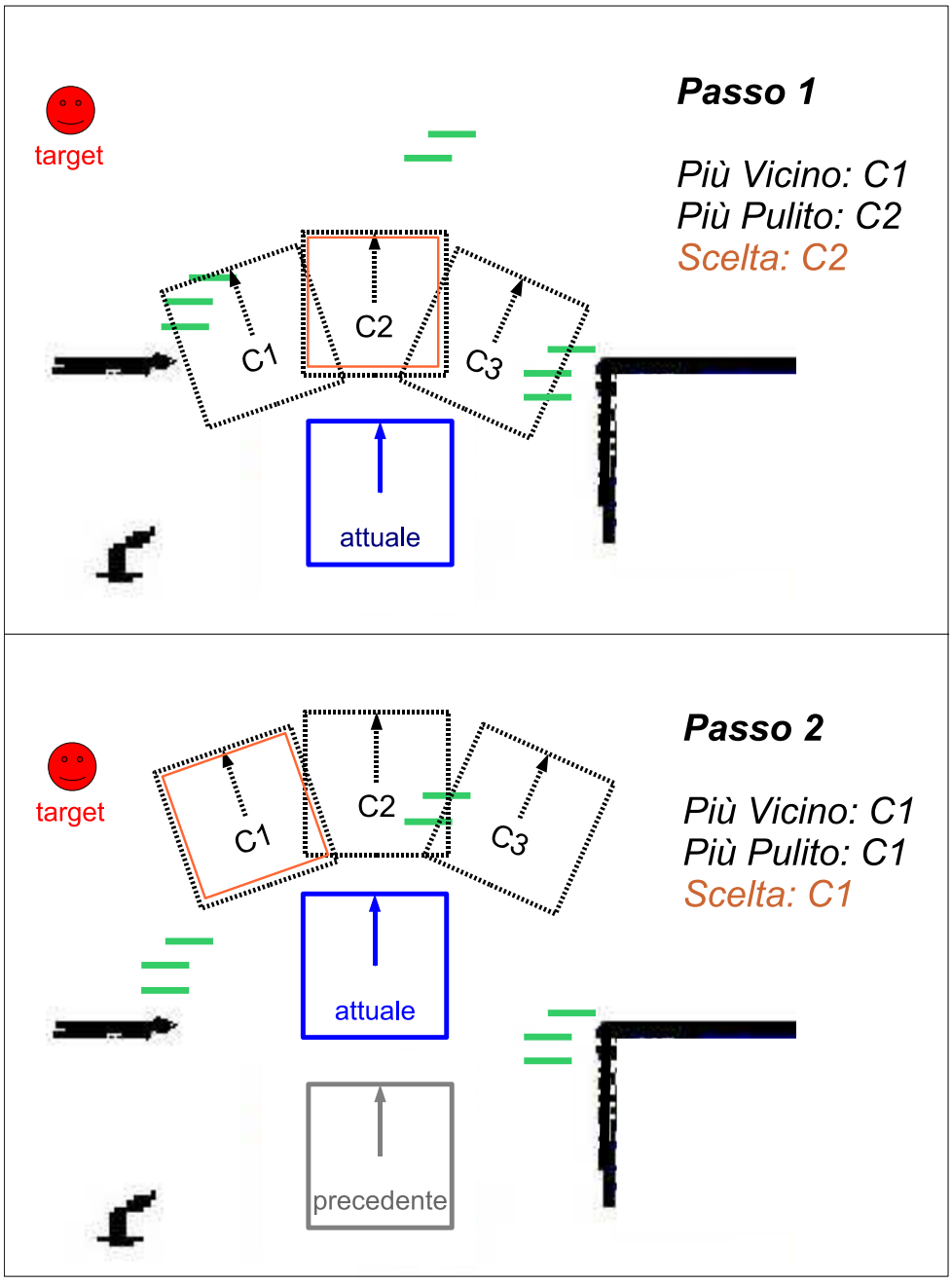
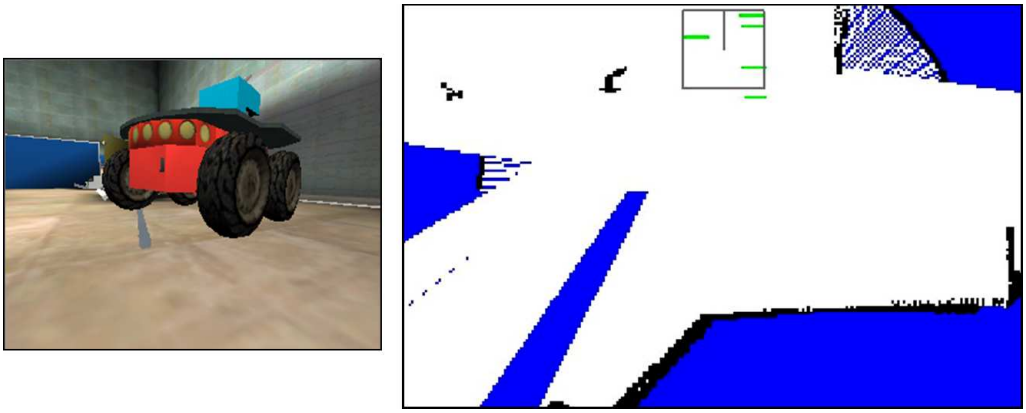


Figura 6.3: Tracciamento della difficoltà di attraversamento di un tubo



Dai test successivi è emerso un buon comportamento del software, in particolare nelle Arene più impegnative.

Le difficoltà incontrate nell'Arena Gialla, corrispondenti a piccoli oggetti isolati, sono state minime e le estensioni al *Sottosistema di Pianificazione del Moto* sono state attivate due sole volte, in corrispondenza di due tubi. Nella Figura 6.3 è mostrato il tracciamento della difficoltà di attraversamento di un tubo, tramite il disegno di diversi segmenti di colore verde in corrispondenza dei punti di attrito tra l'oggetto e le ruote del robot. Sono presenti segmenti in posizioni diverse in quanto durante l'attraversamento il robot ha mosso più volte il tubo.

Nell'Arena Arancione è stata testata l'affidabilità del software in una zona il cui terreno era cosparso uniformemente di fogli di carta, che facevano perdere aderenza alle ruote del robot, con conseguenti slittamenti. Le difficoltà di attraversamento di tale zona sono state correttamente rilevate, ma la mancanza di superfici libere dalla carta non ha consentito la pianificazione di un percorso alternativo.

L'Arena Rossa, che costituiva il banco di prova più impegnativo, è quella in cui l'applicazione si è dimostrata più utile. L'Arena era disseminata di macerie, tubi e lamiere, il cui attraversamento ha portato ad errori di

localizzazione e mapping. Il tracciamento ottenuto nel test dell'applicazione ha permesso una navigazione più sicura e migliorato l'affidabilità delle letture dello scanner laser, consentendo la generazione di una mappa più accurata dell'ambiente.

6.4 Sintesi del lavoro svolto

Lo sviluppo dell'applicazione per la *pianificazione del moto su terreno sconnesso*, in conclusione, ha coinvolto diversi aspetti dell'esplorazione autonoma di un'area, dalla modalità di navigazione alla rappresentazione interna dell'ambiente.

L'interpretazione dei dati relativi alle difficoltà incontrate durante la navigazione ha permesso di ricavare una mappa della qualità del terreno nelle zone esplorate. Tale rappresentazione è stata utilizzata, insieme alle informazioni canoniche sugli ostacoli dell'ambiente, per ottimizzare la costruzione dell'albero di navigazione, allo scopo di evitare le zone più sconnesse, a beneficio di una maggiore stabilità del robot e delle successive letture dei sensori.

Riguardo ai possibili sviluppi futuri dell'applicazione realizzata, un miglioramento potrebbe consistere nel tentativo di suddividere l'ambiente esplorato in sezioni, di estensione variabile, caratterizzate da livelli omogenei di difficoltà nel cammino.

In questo modo sarebbe possibile impostare il *Sottosistema di Pianificazione del Moto* adattando il criterio di costruzione dell'albero di navigazione a seconda dell'*asperità media* della zona occupata.

Capitolo 7

Conclusioni e sviluppi futuri

7.1 Sintesi dei risultati

Nell'arco del mio lavoro con UsarSim ho apprezzato tutti i vantaggi che un simulatore 3D ad alto realismo può offrire nello sviluppo di applicazioni per robot mobili.

La fase iniziale del mio lavoro ha previsto la realizzazione di un'interfaccia tra il simulatore UsarSim e l'Spqr-Rdk (la piattaforma di sviluppo robotico del DIS), il cui primo passo è stato la modellazione e configurazione in UsarSim del robot mobile utilizzato nel laboratorio.

In seguito ho dotato l'Spqr-Rdk del modulo *UsarRobotTask*, in grado di dialogare in tempo reale col robot riprodotto nell'ambiente simulato, consentendone l'utilizzo dei motori e dei principali sensori (laser, sonar, telecamera con sistema pan-tilt-zoom). Allo scopo di completare la simulazione del robot, ho realizzato in UsarSim un meccanismo in grado di emulare una coppia di telecamere stereo, così da poter utilizzare tecniche di stereovisione.

Ho utilizzato il simulatore come banco di prova per il sensore Swiss Camera, realizzandone un modello e testandone in anteprima le qualità, come studio preliminare di fattibilità nell'ottica dell'acquisto del sensore reale da parte del dipartimento. Ho presentato alla comunità di sviluppo

di UsarSim tutte le estensioni realizzate, ora presenti nella release attuale del simulatore.

Completata l'interfaccia tra UsarSim e l'Spqr-Rdk, ho eseguito alcuni test di validazione, volti a verificare il livello di realismo e di integrazione dei sensori nell'ambiente simulato.

Assicurata la stabilità dell'interfaccia realizzata, mi sono dedicato alla realizzazione di alcune applicazioni, utilizzando nell'intero arco di sviluppo UsarSim come ambiente privilegiato di test.

Una prima applicazione ha riguardato la realizzazione del *Sottositema di Recupero dagli Stalli* (SRS), che prevede il tracciamento degli stalli del robot dovuti alla presenza di ostacoli invisibili ai sensori tradizionalmente utilizzati nel processo di mapping (scanner laser, sonar). In mancanza di un sistema di tracciamento, tali ostacoli possono portare il robot in una situazione di stallo permanente, nel tentativo ripetuto di compiere un movimento fallace.

Il tracciamento di tali ostacoli è stato ottenuto tramite l'osservazione dello scarto tra la velocità attuale del robot (stimata in tempo reale) e la velocità desiderata dal sistema di navigazione (autonomo o teleoperato). Su questa base, in caso d'impatto, viene ricostruita la posizione dell'ostacolo invisibile urtato, e segnalata sulla mappa interna del robot. Come ultimo passo, viene inviato un avvertimento al *Sottosistema di Pianificazione del Moto* dell'Spqr-Rdk, che esegue una *ripianificazione veloce* per evitare l'ostacolo ed uscire dalla situazione di stallo.

Lo sviluppo dell'applicazione con UsarSim mi ha permesso di minimizzare i rischi di urti dannosi per il robot reale e di risparmiare tempo nello sviluppo, grazie alla disponibilità continua di un ambiente simulato rapidamente riconfigurabile per test specifici.

Una seconda applicazione realizzata con l'ausilio di UsarSim ha riguardato la pianificazione del moto su terreno sconnesso, allo scopo di evitare le zone che presentino una maggiore difficoltà di attraversamento. Ho rea-

lizzato l'applicazione estendendo le funzionalità del *SRS* e del *Sottosistema di Pianificazione del Moto.*

La tecnica implementata prevede il tracciamento delle asperità del terreno durante la navigazione dell'ambiente, tramite controlli sulla velocità attuale operati nel SRS. Il pianificatore del moto utilizza tali informazioni nel processo di creazione dell'albero di navigazione, tramite una procedura semi-randomica che seleziona i nodi dell'albero, cioè le tappe intermedie, in base alla distanza dalla destinazione ed alla qualità del terreno. Lo sviluppo dell'applicazione è stato semplificato dalle ripetute verifiche in ambienti simulati di difficoltà variabile, utilizzando uno spettro di possibilità di test (rampe, scale, macerie, pozzi, tubi, terreno franato, vetri, lamiere) difficilmente raggiungibile negli ambienti riproducibili in laboratorio.

In conclusione, in questi pochi mesi di sviluppo con UsarSim ho scoperto tutti i vantaggi nel lavorare in un ambiente simulato realisticamente in 3D ed apprezzato la facilità con cui è stato possibile modificarlo ed estenderlo per venire incontro ai bisogni di un gruppo di sviluppo.

UsarSim mi ha permesso di riprodurre con facilità il robot utilizzato in laboratorio e di simulare realisticamente le sue interazioni in ambienti altamente destrutturati. Infine mi ha permesso di migliorare la pianificazione del moto su terreno sconnesso e di affrontare l'annoso problema degli ostacoli invisibili ai sensori, minimizzando il numero di urti del robot reale e soprattutto i tempi di sviluppo.

7.2 Possibili sviluppi futuri

UsarSim è un simulatore recente e la comunità che lo utilizza è sufficientemente ampia da garantire molteplici sviluppi ed estensioni, soprattutto nella modellazione delle dinamiche fisiche. Gli sviluppi previsti per la prossima versione di UsarSim sono descritti nel paragrafo 2.4.4.

Considerata l'esperienza maturata personalmente, vedo possibilità di

miglioramento nella simulazione dei sensori basati su raggi infrarossi, tra cui lo scanner laser e l'IRC. In particolare, sarebbe interessante lavorare sull'interazione realistica tra la radiazione infrarossa e le superfici riflettenti, quali gli specchi.

Riguardo agli scenari multiagente, una limitazione dell'architettura attuale di UsarSim rimane la necessità di eseguire una diversa finestra di *Unreal Client* per acquisire contemporaneamente il feedback video di ogni robot dotato di telecamera.

Nell'architettura attuale del simulatore non è possibile eseguire diverse copie di *Unreal Client* su uno stesso sistema operativo, con la conseguenza che la simulazione di n robot dotati di telecamera necessita l'utilizzo di n diversi PC.

Considerata tale pesante limitazione, una semplice soluzione al problema potrebbe essere rappresentata dalla divisione della finestra di un unico *Unreal Client* in vari riquadri in grado di ospitare il feedback video di ogni telecamera, analogamente a quanto già realizzato (come descritto nella sezione 4.3.1) per rendere possibile la stereovisione.

Appendice A

Codice sorgente

A.1 Specifica del robot simulato

```
class P2AT extends KRobot config(USAR);

var float cachedLeftValue,cachedRightValue;
var rotator cachedCameraRot,initCameraRot;
var bool bGetInitCameraRot;
var config bool bAbsoluteCamera;
var config bool useStereoVision;
var config vector stereoSpacing; //(eye to eye)/2 spacing in UU

// beg::exp
var config float paperError1, paperError2;
var config float paperProb;
// end::exp

simulated function DrawHud(Canvas C)
{
  local int halfSizeX;
  local vector stereoRotated;
  if (useStereoVision) {
```

```
    halfSizeX = C.SizeX/2.0;
    stereoRotated = stereoSpacing >> myCamera.Rotation;
    C.Reset();
    C.Clear(); //Clears frame & Z buffers
    //left Eye:
    C.DrawPortal(0, 0, halfSizeX, C.SizeY, myCamera,
        myCamera.Location - stereoRotated, myCamera.Rotation,
        CameraZoom);
    //Right Eye:
    C.DrawPortal(halfSizeX, 0, halfSizeX, C.SizeY, myCamera,
        myCamera.Location + stereoRotated, myCamera.Rotation,
        CameraZoom);
  }
  super.DrawHud(C);
}

function ProcessCarInput()
{
  local float LeftValue,RightValue;
  local int CameraPan,CameraTilt;
  Super.ProcessCarInput();
  if (!bGetInitCameraRot) {
    if (myCamera!=None)
    initCameraRot = myCamera.Rotation;
    RemoteBot(Controller).LeftThrottle = 0.0;
    RemoteBot(Controller).RightThrottle = 0.0;
    RemoteBot(Controller).myCameraRotation = rot(0,0,0);
    bGetInitCameraRot = true;
  }
  LeftValue = RemoteBot(Controller).LeftThrottle*850;
  RightValue = RemoteBot(Controller).RightThrottle*850;
  if (cachedLeftValue!=LeftValue) {
    JointsControl[1].state = 1; // new command
    JointsControl[1].order = 1;
```

```
    JointsControl[1].value = LeftValue;
    JointsControl[3].state = 1; // new command
    JointsControl[3].order = 1;
    JointsControl[3].value = LeftValue;
    bNewCommand = true;
    cachedLeftValue=LeftValue;
  }
  if (cachedRightValue!=RightValue) {
    JointsControl[0].state = 1; // new command
    JointsControl[0].order = 1;
    JointsControl[0].value = RightValue;
    JointsControl[2].state = 1; // new command
    JointsControl[2].order = 1;
    JointsControl[2].value = RightValue;
    bNewCommand = true;
    cachedRightValue=RightValue;
  }
  if (myCamera!=None) {
    if (RemoteBot(Controller).myCameraRotation.roll!=0) return;
    RemoteBot(Controller).myCameraRotation.roll = -1;
    if (RemoteBot(Controller).myCameraRotationOrder==1) {
      CameraTilt = RemoteBot(Controller).myCameraRotation.Pitch;
      CameraPan = RemoteBot(Controller).myCameraRotation.Yaw;
    } else if (!bAbsoluteCamera) {
      CameraTilt = -RemoteBot(Controller).myCameraRotation.Pitch;
      CameraPan = RemoteBot(Controller).myCameraRotation.Yaw;
    } else {
      CameraTilt = myCamera.Rotation.Pitch
          - initCameraRot.Pitch - Rotation.Pitch
          - RemoteBot(Controller).myCameraRotation.Pitch;
      CameraPan = RemoteBot(Controller).myCameraRotation.Yaw
          - myCamera.Rotation.Yaw
          + initCameraRot.Yaw + Rotation.Yaw;
      if (CameraTilt>65535) CameraTilt-=65535;
```

```
        else if (CameraTilt<-65535) CameraTilt+=65535;
        if (CameraTilt>32768) CameraTilt-=65535;
        else if (CameraTilt<-32768) CameraTilt+=65535;
        if (CameraPan>65535) CameraPan-=65535;
        else if (CameraPan<-65535) CameraPan+=65535;
        if (CameraPan>32768) CameraPan-=65535;
        else if (CameraPan<-32768) CameraPan+=65535;
      }
      if (CameraPan!=0
      || RemoteBot(Controller).myCameraRotationOrder==1) {
        JointsControl[5].state = 1;
        JointsControl[5].order =
            RemoteBot(Controller).myCameraRotationOrder;
        JointsControl[5].value = CameraPan;
      }
      if (JointsControl[5].order==1)
        PanSpeed = CameraPan;
      else
        PanSpeed = MotorSpeed;
      if (CameraTilt!=0
      || RemoteBot(Controller).myCameraRotationOrder==1) {
        JointsControl[6].state = 1;
        JointsControl[6].order =
            RemoteBot(Controller).myCameraRotationOrder;
        JointsControl[6].value = CameraTilt;
      }
      if (JointsControl[6].order==1)
        TiltSpeed = CameraTilt;
      else
        TiltSpeed = MotorSpeed;
      bNewCommand = true;
    }
  }
```

```
simulated function Tick(float Delta)
{
  Super.Tick(Delta);
  if ( PhysicsVolume.LocationName=="Papers"
  && GetStateName()!='CrossPapers')
    GotoState('CrossPapers','Begin');
}

// beg::exp
state CrossPapers{
  function BeginState()
  {
    log("Enter state: CrossPapers");
  }
  function damp()
  {
    if (FRand()<paperProb)
      KCarWheelJoint(Joints[0]).KMaxSpeed = cachedRightValue
          + RandRange(paperError1,paperError2)*cachedRightValue;
    if (FRand()<paperProb)
      KCarWheelJoint(Joints[1]).KMaxSpeed = cachedLeftValue
          + RandRange(paperError1,paperError2)*cachedLeftValue;
    if (FRand()<paperProb)
      KCarWheelJoint(Joints[2]).KMaxSpeed = cachedRightValue
          + RandRange(paperError1,paperError2)*cachedRightValue;
    if (FRand()<paperProb)
      KCarWheelJoint(Joints[3]).KMaxSpeed = cachedLeftValue
          + RandRange(paperError1,paperError2)*cachedLeftValue;
  }
  function EndState()
  {
    log("Leave state: CrossPapers");
  }
  Begin:
```

```
    damp();
    Sleep(0.2);
    if ( PhysicsVolume.LocationName=="Papers") goto('Begin');
      GotoState('Rover');
}
// end::exp

defaultproperties
{
  bDebug=false
  StaticMesh=StaticMesh'USAR_Meshes.Robot.P2AT'
  DrawScale=1.0
  DrawScale3D=(X=1.0,Y=1.0,Z=1.0)
  TireSlipRate=0.00700
  useStereoVision = false
  stereoSpacing=(Y=12) //4 cm: (eye to eye)/2 spacing in UU


  // beg::exp
  paperProb=0.3
  paperError1=-0.05
  paperError2=0.05
  // end::exp

  Begin Object Class=KarmaParamsRBFull Name=KParams0
    KInertiaTensor(0)=20.000000
    KInertiaTensor(3)=30.000000
    KInertiaTensor(5)=48.000000
    KCOMOffset=(X=0.000000,Z=-15.000000)
    KLinearDamping=0.000000
    KAngularDamping=0.000000
    KStartEnabled=True
    bHighDetailOnly=False
    bClientOnly=False
```

```
    bKDoubleTickRate=True
    KFriction=1.600000
    Name="KParams0"
  End Object
  KParams=KarmaParamsRBFull'USARBot.P2AT.KParams0'
  Begin Object Class=KarmaParamsRBFull Name=KarmaParams2
    KInertiaTensor(0)=1.8
    KInertiaTensor(1)=0
    KInertiaTensor(2)=0
    KInertiaTensor(3)=1.8
    KInertiaTensor(4)=0
    KInertiaTensor(5)=1.8
    KAngularDamping=0
    KLinearDamping=0
    bHighDetailOnly=False
    bClientOnly=False
    bKDoubleTickRate=True
    Name="KarmaParams2"
  End Object

  JointParts(0)=(PartName="LeftFWheel",PartClass=class'USARBot.KDTire',
    PartStaticMesh=StaticMesh'BulldogMeshes.Simple.S_RearWheel',
    DrawScale=0.1,KParams=KarmaParams'USARBot.P2AT.KarmaParams2',
    DrawScale3D=(X=1.0,Y=0.6,Z=1.0),bSteeringLocked=True,
    bSuspensionLocked=true,Parent="",JointClass=class'KCarWheelJoint',
    ParentPos=(X=20,Y=35.0,Z=-10.0),ParentRot=(Yaw=16384),
    ParentAxis=(Z=1.0),SelfPos=(Z=0.0),SelfAxis=(Z=1.0))
  JointParts(1)=(PartName="RightFWheel",PartClass=class'USARBot.KDTire',
    PartStaticMesh=StaticMesh'BulldogMeshes.Simple.S_RearWheel',
    DrawScale=0.1,KParams=KarmaParams'USARBot.P2AT.KarmaParams2',
    DrawScale3D=(X=1.0,Y=0.6,Z=1.0),bSteeringLocked=True,
    bSuspensionLocked=true,Parent="",JointClass=class'KCarWheelJoint',
    ParentPos=(X=20,Y=-35.0,Z=-10.0),ParentRot=(Yaw=49152),
    ParentAxis=(Z=1.0),SelfPos=(Z=0.0),SelfAxis=(Z=1.0))
```

```
    JointParts(2)=(PartName="RearWheel",PartClass=class'USARBot.KDTire',
      PartStaticMesh=StaticMesh'BulldogMeshes.Simple.S_RearWheel',
      DrawScale=0.03,KParams=KarmaParams'USARBot.P2AT.KarmaParams2',
      DrawScale3D=(X=1.0,Y=0.6,Z=1.0),bSteeringLocked=True,
      bSuspensionLocked=true,Parent="",JointClass=class'KCarWheelJoint',
      ParentPos=(X=-35,Y=0.0,Z=-15.0),ParentRot=(Yaw=16384),
      ParentAxis=(Z=1.0),SelfPos=(Z=0.0),SelfAxis=(Z=1.0))
}
```

A.2 Configurazione del robot simulato

```
[USARBot.P2AT]
paperProb=0.5
paperError1=-0.2
paperError2=0.2
bAbsoluteCamera=True
useStereoVision=True
bDisplayTeamBeacon=False
bDebug=False
ChassisMass=1.000000
MotorTorque=40.0

JointParts=(PartName="RightFWheel",PartClass=class'USARBot.P2ATTire',
  PartStaticMesh=StaticMesh'BulldogMeshes.Simple.S_RearWheel',
  DrawScale=0.2,KParams=KarmaParams'USARBot.P2AT.KarmaParams2',
  DrawScale3D=(X=1.0,Y=0.55,Z=1.0),bSteeringLocked=True,
  bSuspensionLocked=true,Parent="",
  JointClass=class'KCarWheelJoint',ParentPos=(Y=51,X=33.0,Z=-28.0),
  ParentRot=(Yaw=0),ParentAxis=(Z=1.0),ParentAxis2=(Y=1.0),
  SelfPos=(Z=0.0),SelfAxis=(Z=1.0),SelfAxis2=(Y=1.0))
JointParts=(PartName="LeftFWheel",PartClass=class'USARBot.P2ATTire',
  PartStaticMesh=StaticMesh'BulldogMeshes.Simple.S_RearWheel',
  DrawScale=0.2,KParams=KarmaParams'USARBot.P2AT.KarmaParams2',
```

```
    DrawScale3D=(X=1.0,Y=0.55,Z=1.0),bSteeringLocked=True,
    bSuspensionLocked=true,Parent="",
    JointClass=class'KCarWheelJoint',ParentPos=(Y=-51,X=33.0,Z=-28.0),
    ParentRot=(Yaw=32768),ParentAxis=(Z=1.0),ParentAxis2=(Y=1.0),
    SelfPos=(Z=0.0),SelfAxis=(Z=1.0),SelfAxis2=(Y=1.0))
  JointParts=(PartName="RightRWheel",PartClass=class'USARBot.P2ATTire',
    PartStaticMesh=StaticMesh'BulldogMeshes.Simple.S_RearWheel',
    DrawScale=0.2,KParams=KarmaParams'USARBot.P2AT.KarmaParams2',
    DrawScale3D=(X=1.0,Y=0.55,Z=1.0),bSteeringLocked=True,
    bSuspensionLocked=true,Parent="",JointClass=class'KCarWheelJoint',
    ParentPos=(Y=51,X=-33.0,Z=-28.0),ParentRot=(Yaw=0),
    ParentAxis=(Z=1.0),ParentAxis2=(Y=1.0),SelfPos=(Z=0.0),
    SelfAxis=(Z=1.0),SelfAxis2=(Y=1.0))
  JointParts=(PartName="LeftRWheel",PartClass=class'USARBot.P2ATTire',
    PartStaticMesh=StaticMesh'BulldogMeshes.Simple.S_RearWheel',
    DrawScale=0.2,KParams=KarmaParams'USARBot.P2AT.KarmaParams2',
    DrawScale3D=(X=1.0,Y=0.55,Z=1.0),bSteeringLocked=True,
    bSuspensionLocked=true,Parent="",JointClass=class'KCarWheelJoint',
    ParentPos=(Y=-51,X=-33.0,Z=-28.0),ParentRot=(Yaw=32768),
    ParentAxis=(Z=1.0),ParentAxis2=(Y=1.0),SelfPos=(Z=0.0),
    SelfAxis=(Z=1.0),SelfAxis2=(Y=1.0))

  ;pan-tilt system
  JointParts=(PartName="CameraBase",PartClass=class'USARBot.CameraBase',
    PartStaticMesh=StaticMesh'USAR_Meshes.Robot.CameraBase',
    DrawScale=1.0,KParams=KarmaParams'USARBot.P2AT.KarmaParams2',
    DrawScale3D=(X=1.0,Y=1.0,Z=1.0),bSteeringLocked=True,
    bSuspensionLocked=true,BrakeTorque=25.0,Parent="",
    JointClass=class'KCarWheelJoint',ParentPos=(Y=0.0,X=32.0,Z=54.0),
    ParentRot=(Yaw=32768),ParentAxis=(Z=1.0),ParentAxis2=(X=1.0),
    SelfPos=(Z=0.0),SelfAxis=(Z=1.0),SelfAxis2=(X=1.0))
  JointParts=(PartName="CameraPan",PartClass=class'USARBot.CameraPan',
    PartStaticMesh=StaticMesh'USAR_Meshes.Robot.CameraPanFrame',
    DrawScale=1.0,KParams=KarmaParams'USARBot.P2AT.KarmaParams2',
```

```
    DrawScale3D=(X=1.0,Y=1.0,Z=1.0),bSteeringLocked=True,
    bSuspensionLocked=true,BrakeTorque=5.0,Parent="",
    JointClass=class'KCarWheelJoint',ParentPos=(Y=0.0,X=32.0,Z=67.0),
    ParentRot=(Yaw=32768),ParentAxis=(Y=1.0),ParentAxis2=(Z=1.0),
    SelfPos=(Z=7.0),SelfAxis=(Y=1.0),SelfAxis2=(Z=1.0))
JointParts=(PartName="CameraTilt",PartClass=class'USARBot.CameraTilt',
    PartStaticMesh=StaticMesh'USAR_Meshes.Robot.CameraTiltFrame',
    DrawScale=1.0,KParams=KarmaParams'USARBot.P2AT.KarmaParams2',
    DrawScale3D=(X=1.0,Y=1.0,Z=1.0),bSteeringLocked=True,
    bSuspensionLocked=true,BrakeTorque=5.0,Parent="CameraPan",
    JointClass=class'KCarWheelJoint',ParentPos=(X=0.0,Y=0.0,Z=18.0),
    ParentRot=(Yaw=32768),ParentAxis=(Z=1.0),ParentAxis2=(Y=1.0),
    SelfPos=(X=0.0,Y=0.0,Z=0.0),SelfAxis=(Z=1.0),SelfAxis2=(Y=1.0))

Camera=(ItemClass=class'USARBot.RobotCamera',ItemName="Camera",
    Parent="CameraTilt",Position=(Y=0,X=20,Z=-16),
    Direction=(Pitch=0,Yaw=0,Roll=0))
HeadLight=(ItemClass=class'USARBot.USARHeadLight',
    ItemName="Headlight",Parent="CameraTilt",
    Position=(Y=5,X=20,Z=-16),Direction=(Pitch=-2000,Yaw=0,Roll=0))
HeadLight=(ItemClass=class'USARBot.USARHeadLight',
    ItemName="Headlight",Parent="CameraTilt",
    Position=(Y=-5,X=20,Z=-16),Direction=(Pitch=-2000,Yaw=0,Roll=0))
Sensors=(ItemClass=class'USARBot.OdometrySensor',
    ItemName="Odometry",Position=(X=0,Y=0,Z=0),
    Direction=(Pitch=0,Yaw=0,Roll=0))

;Range Sensor dismissed, Sonar Sensor (more accurate) included
Sensors=(ItemClass=class'USARBot.SonarSensor',ItemName="F1",
    Position=(X=36.25,Y=-32.50,Z=0),
    Direction=(Pitch=0,Yaw=-16384,Roll=0))
Sensors=(ItemClass=class'USARBot.SonarSensor',ItemName="F2",
    Position=(X=46.25,Y=-28.75,Z=0),
    Direction=(Pitch=0,Yaw=-9102,Roll=0))
```

```
Sensors=(ItemClass=class'USARBot.SonarSensor',ItemName="F3",
  Position=(X=55.00,Y=-20.00,Z=0),
  Direction=(Pitch=0,Yaw=-5461,Roll=0))
Sensors=(ItemClass=class'USARBot.SonarSensor',ItemName="F4",
  Position=(X=60.00,Y=-6.25,Z=0),
  Direction=(Pitch=0,Yaw=-1820,Roll=0))
Sensors=(ItemClass=class'USARBot.SonarSensor',ItemName="F5",
  Position=(X=60.00,Y=6.25,Z=0),
  Direction=(Pitch=0,Yaw=1820,Roll=0))
Sensors=(ItemClass=class'USARBot.SonarSensor',ItemName="F6",
  Position=(X=55.00,Y=20.00,Z=0),
  Direction=(Pitch=0,Yaw=5461,Roll=0))
Sensors=(ItemClass=class'USARBot.SonarSensor',ItemName="F7",
  Position=(X=46.25,Y=28.75,Z=0),
  Direction=(Pitch=0,Yaw=9120,Roll=0))
Sensors=(ItemClass=class'USARBot.SonarSensor',ItemName="F8",
  Position=(X=36.25,Y=32.50,Z=0),
  Direction=(Pitch=0,Yaw=16384,Roll=0))

Sensors=(ItemClass=class'USARBot.SonarSensor',ItemName="R1",
  Position=(X=-36.25,Y=32.50,Z=0),
  Direction=(Pitch=0,Yaw=16384,Roll=0))
Sensors=(ItemClass=class'USARBot.SonarSensor',ItemName="R2",
  Position=(X=-46.25,Y=28.75,Z=0),
  Direction=(Pitch=0,Yaw=23666,Roll=0))
Sensors=(ItemClass=class'USARBot.SonarSensor',ItemName="R3",
  Position=(X=-55.00,Y=20.00,Z=0),
  Direction=(Pitch=0,Yaw=27307,Roll=0))
Sensors=(ItemClass=class'USARBot.SonarSensor',ItemName="R4",
  Position=(X=-60.00,Y=6.25,Z=0),
  Direction=(Pitch=0,Yaw=30948,Roll=0))
Sensors=(ItemClass=class'USARBot.SonarSensor',ItemName="R5",
  Position=(X=-60.00,Y=-6.25,Z=0),
  Direction=(Pitch=0,Yaw=-30948,Roll=0))
```

```
Sensors=(ItemClass=class'USARBot.SonarSensor',ItemName="R6",
  Position=(X=-55.00,Y=-20.00,Z=0),
  Direction=(Pitch=0,Yaw=-27307,Roll=0))
Sensors=(ItemClass=class'USARBot.SonarSensor',ItemName="R7",
  Position=(X=-46.25,Y=-28.75,Z=0),
  Direction=(Pitch=0,Yaw=-23666,Roll=0))
Sensors=(ItemClass=class'USARBot.SonarSensor',ItemName="R8",
  Position=(X=-36.25,Y=-32.50,Z=0),
  Direction=(Pitch=0,Yaw=-16384,Roll=0))

Sensors=(ItemClass=class'USARBot.SICKLMS',ItemName="Scanner1",
  Position=(X=36,Y=0,Z=23),Direction=(Pitch=0,Yaw=0,Roll=0))
```

A.3 Range Scanner Sensor

```
//class RangeScanner extends RangeSensor config (USAR);
class RangeScanner extends IRSensor config (USAR);

var config float ScanInterval;
var config float Resolution;
var config int ScanFov;
var config bool bYaw;
var config bool bPitch;
var string rangeData;
var RemoteBot rBot;

simulated function PostNetBeginPlay()
{
  Super.PostNetBeginPlay();
  rangeData="";
  if (ScanInterval>=0.1)
    SetTimer(ScanInterval,true);
}
```

```
function timer()
{
  local int i;
  local float range;
  rangeData="";
  curRot = base.Rotation + myDirection;
  for (i=ScanFov/2;i>-ScanFov/2;i-=Resolution)
  {
    if (bYaw)
      curRot.Yaw = Rotation.Yaw + i;
    if (bPitch)
      curRot.Pitch = Rotation.Pitch + i;
    range = GetRange();
    if (rangeData == "")
      rangeData = ""$range;
    else
      rangeData = rangeData$","$range;
  }
}


function String GetData()
{
  local string outstring;
  if (rBot == None)
    rBot = RemoteBot(platform.Controller);
  if (rBot.RangeScan && rBot.senName==SenName)
  {
    timer();
    rBot.RangeScan = false; //reset command
  }
  else if (rangeData == "")
    return "";
```

```
  outstring = "SEN "$"{Type "$SenType$"} "
      $"{Name "$SenName$"} "
      $"{Time "$Level.TimeSeconds$"} "
      //$"{Location "$Location$"} "
      //$"{Rotation "$(base.Rotation + myDirection)$"} "
      $"{Resolution "$Resolution$"} "
      $"{FOV "$ScanFov$"} "
      $"{Range "$rangeData$"}";
  rangeData = "";
  return outstring;
}

function String GetConf()
{
  local string outstring;
  outstring = Super.GetConf();
  outstring = outstring@"{Resolution "$Resolution$"} {Fov "$ScanFov$"}";
  outstring = outstring@"{Paning "$bYaw$"} {Tilting "$bPitch$"}";
  return outstring;
}

defaultproperties
{
  bHardAttach=true
  SenType="RangeScanner"
  MaxRange=1000
  MinRange=10
  ScanInterval=0.0
  Resolution=1000
  ScanFov=32768
  bYaw=false
  bPitch=false
  OutputCurve=(Points=((InVal=0,OutVal=0),(InVal=1000,OutVal=1000)))
}
```

Bibliografia

[1] Jacoff, et al, *NIST Reference Test Facility for Autonomous Mobile Robots for Urban Search and Rescue*, 2001

[2] J. Wang, M. Lewis, J. Gennari, *USAR: A Game-Based Simulation for Teleoperation*, Proc. 47th Ann. Meeting Human Factors and Ergonomics Soc., 2003

[3] J. Wang, M. Lewis, J. Gennari, *A Game Engine Based Simulation of the NIST USAR Arenas*, Proc. 2003 Winter Simulation Conf., Winter Simulation Board, 2003

[4] S. Carpin, A. Birk, M. Lewis, A. Jacoff, *High fidelity tools for rescue robotics: results and perspectives*, RoboCup International Symposium 2005, Osaka (Japan), 2005

[5] J. Wang, M. Lewis, M. Koes, S. Carpin, *Validating USARsim for use in HRI Research*, Proc. of the Human Factors And Ergonomics Society 49th Annual Meeting 2005, pp. 457-461, 2005

[6] J.J Leonard, H.F. Durrant-Whyte, *Directed sonar sensing for mobile robot navigation*, Kluwer Academic Publishers, 1992

[7] O. Michel, Cyberbotics Ltd, *WebotsTM: Professional Mobile Robot Simulation*, International Journal of Advanced Robotic Systems, Volume 1 Number 1, pp. 39-42, 2004, ISSN 1729-8806

[8] T. Laue, K. Spiess, T. Röfer, *SimRobot - A General Physical Robot Simulator and Its Application in RoboCup*, in RoboCup 2005: Robot Soccer World Cup IX, Lecture Notes in Artificial Intelligence, 2005

[9] A. Farinelli, G. Grisetti, L. Iocchi, *SPQR-RDK: a modular framework for programming mobile robots*, in Proc. of Int. RoboCup Symposium 2004, pp. 653-660, 2005

[10] S. Bahadori, D. Calisi, A. Censi, A. Farinelli, G. Grisetti, L. Iocchi, D. Nardi, *Intelligent Systems for Search and Rescue,* in Proc. of IROS Workshop "Urban search and rescue: from Robocup to real world applications", 2004

[11] B. Buettgen, *High-Integrated Solid-State 3D-Camera - Principle, Limitations and Applications*, CSEM Centre Suisse d'Electronique et de Microtechnique (Presentazione), 2004

[12] G. Grisetti, C. Stanchniss, W. Burgard, *Improving Grid-based SLAM with Rao-Blackwellized Particle Filters by Adaptive Proposals and Selective Resampling*, In. Proc. of the IEEE International Conference on Robotics and Automation (ICRA), 2005

[13] L. Marchionni, *Costruzione di mappe per robot mobili con sensori eterogenei* (Tesi di Laurea), 2005

[14] D. Calisi, *Pianificazione del percorso per un robot tramite mappe topologiche probabilistiche e adattamento della traiettoria* (Tesi di Laurea), 2004

[15] J.L. Jones, A.M. Flynn, *Mobile Robots - Inspiration to Implementation*, A K Peters Ltd., Wellesley, Massachusetts, 1993

[16] G.R. Scholz, C.D. Rahn, *Profile Sensing with an Artuated Whisker*, IEEE Transactions on Robotics and Automation, Vol. 20, No. 1, pp. 124-127, 2004

[17] W. Burgard, A.B., Cremers, D. Fox, D. Hahnel, G. Lakemeyer, D. Schulz,W. Steiner, and S. Thrun, *The interactive museum tour-guide robot*, In Proceedings of the AAAI Fifteenth National Conference on Artificial Intelligence, 1998

[18] D. Fox, W. Burgard, S. Thrun, A. Cremers, *A hybrid collision avoidance method for mobile robots*, In Proc. IEEE International Conference on Robotics and Automation (ICRA), 1998

[19] G. Oriolo, *Pianificazione del moto* (Dispense), 1993

[20] L. Kavraki, J. Latombe, *Probabilistic roadmaps for robot path planning*, In Practical Motion Planning in Robotics: Current Approaches and Future Challenges, pp. 33-53, 1998

[21] Daniele Calisi, Alessandro Farinelli, Luca Iocchi, Daniele Nardi, *Autonomous Navigation and Exploration in a Rescue Environment,* RoboCup Symposium 2004

[22] S. LaValle, J. Kuner, *Randomized kinodynamic planning*, In Proc. IEEE International Conference on Robotics and Automation (ICRA), pp. 473-479, 1999

[23] J. Kuner, S. LaValle, *Rrt-connect: An efficient approach to single-query path planning*, In Proc. IEEE International Conference on Robotics and Automation (ICRA), 2000

Siti web consultati

Siti web relativi ad UsarSim:

- Epic Games, `http://www.epicgames.com`
- Gamebots, `http://www.planetunreal.com/gamebots`
- Unreal Developer Network, `http://udn.epicgames.com/Main/WebHome`
- Unreal Engine, `http://unreal.epicgames.com`
- Unreal Engine 2 Runtime, `http://edn.epicgames.com/Two/UnrealEngine2Runtime22262001`
- Unreal Tournament 2003, `http://www.unrealtournament2003.com`
- Unreal Tournament 2004, `http://www.unrealtournament2004.com`
- UsarSim Latest Release, `http://usl.sis.pitt.edu/wjj/USAR/Release/`
- UsarSim Official Download Page, `http://usl.sis.pitt.edu/ulab/usarsim_download_page.html`
- Virtual Robots, Università di Brema, `http://www.faculty.iu-bremen.de/carpin/VirtualRobots/`

Altri simulatori orientati alla robotica:

- Gazebo, Player e Stage, `http://playerstage.sourgeforge.net`

- ODE - Open Document Engine, `http://www.ode.org`
- Pyro, `http://pyrorobotics.org`
- RobotCafe Simulators, `http://www.robotcafe.com/dir/Software/Simulators/`
- SimRobot, `http://www.informatik.uni-bremen.de/simrobot/index_e.html`
- UchilSim, `http://www.robocup.cl/uchilsim/`
- ÜberSim, `http://www.cs.cmu.edu/~robosoccer/ubersim/`
- Webots, `http://www.cyberobotics.com`

Siti web relativi alla piattaforma Spqr-Rdk:

- DIS - Dipartimento di Informatica e Sistemistica, `http://www.dis.uniroma1.it`
- SIED - Sistemi Intelligenti per le Emergenze e la Difesa civile, `http://www.dis.uniroma1.it/~multirob/sied`
- Spqr-Rdk, `http://www.dis.uniroma1.it/~spqr/Rescue.htm`

Altri siti web:

- ActivMedia Robotics, `http://activrobots.com`
- Carneige Mellon University, `http://www.cmu.edu`
- CSEM, Swiss Center for Electronics and Microtechnology, `http://www.csem.ch`
- German Team, `http://www.germanteam.org`
- ISI - Information Science Institute, `http://www.isi.edu`

- NIST Intelligent Systems Division - Performance Metrics and Test Arenas for Autonomous Mobile Robots, `http://robotarenas.nist.gov/competitions.htm`
- NSF - National Science Foundation, `http://www.nsf.org`
- RoboCup, `http://www.robocup2005.org`
- RoboCup Rescue-Virtual Mailing List Page, `https://mailman.cc.gatech.edu/mailman/listinfo/robocup-rescue-v`
- RoSiML - Robot Simulation Markup Language, `http://www.tzi.de/spprobocup/RoSiML.html`
- SRI - Stanford Research Institute, `http://www.ai.sri.com`

Elenco degli Algoritmi

Elenco delle Tabelle

Elenco delle Figure

tratto da Dylan Dog n°73, "Tre per zero", pagina 73
per gentile concessione di Sergio Bonelli Editore

Questa tesi è stata compilata utilizzando il software LaTeX2e