UNIVERSITÀ DEGLI STUDI DI PADOVA

FACOLTÀ DI INGEGNERIA

LAUREA SPECIALISTICA IN INGEGNERIA INFORMATICA

Un ambiente integrato per la gestione di laboratori robotici virtuali

Laureando:
Francesco SOLERO

Relatore:
Prof. Carlo FERRARI

Padova, 16 Luglio 2012

Sommario

L'obiettivo di questa tesi è la realizzazione di un ambiente integrato per la predisposizione di laboratori virtuali condivisibili in rete sui quali poter svolgere esperimenti nel campo della robotica.

In questo elaborato vengono descritte le varie fasi di analisi del problema, progettazione e realizzazione che hanno portato allo sviluppo del sistema di simulazione con le caratteristiche richieste. L'organizzazione dei capitoli ripercorre la sequenza delle fasi seguite durante il lavoro di tesi.

Nel capitolo 1 viene fornita una introduzione al concetto di simulazione e il suo utilizzo nel campo della robotica, la descrizione del progetto VLAB seguita da una analisi sullo stato dell'arte costituita da una selezione rappresentativa dei programmi di simulazione per la robotica attualmente a disposizione.

Nel capitolo 2 viene descritta la fase di analisi dei requisiti che ha portato alla definizione delle caratteristiche da realizzare per il sistema di simulazione e alla conseguente fase di progettazione della sua architettura e degli elementi che lo compongono.

Nel capitolo 3 viene descritta la fase di realizzazione vera e propria del sistema progettato nelle fasi precedenti. Vengono descritti in particolare il nucleo del sistema composto dal motore di simulazione, le componenti preposte alla comunicazione, i meccanismi per l'esecuzione dei programmi dei robot e le interfacce previste per l'accesso all'ambiente simulato.

Il capitolo 4 è dedicato alla fase di progettazione del linguaggio di descrizione delle scene e la realizzazione di un opportuno interprete in grado di inserire nella simulazione gli elementi descritti dall'utente. Viene inoltre fornita una descrizione di tutti gli elementi che costituiscono il linguaggio e le regole dalla grammatica da utilizzare per i file descrittivi da

inserire nella simulazione.

Nel capitolo 5 vengono presentati a titolo di riferimento alcuni esempi di codice descrittivo per la rappresentazione di elementi da inserire nella simulazione.

Infine nell'ultimo capitolo vengono tratte le conclusioni sui risultati ottenuti con questo lavoro di tesi, seguite da una breve analisi per delineare la direzione di eventuali sviluppi futuri.

Indice

Capitolo 1

Introduzione

1.1 La simulazione nel campo della robotica

I software di simulazione

La simulazione consiste nella creazione di un modello matematico di un sistema che permetta di elaborare una previsione sull'evoluzione del suo comportamento. La simulazione al computer adotta gli stessi principi, ma prevede che il modello sia creato tramite l'uso di un elaboratore, sfruttandone il potenziale di calcolo e adottando opportune tecniche di risoluzione, rendendo possibile studiare sistemi di dimensioni e complessità che non potrebbero essere analizzati tramite tecniche tradizionali.

Un programma di simulazione è un software progettato per riproporre con un certo grado di fedeltà una situazione, reale o ipotetica, futura o già avvenuta per poter fare analisi, trarre indicazioni, eseguire dei test, effettuare previsioni o addirittura per puro svago dell'utente. I campi di adozione di un simulatore sono tra i più vari e di conseguenza le caratteristiche presentate possono risultare decisamente differenti a seconda dell'ambito di utilizzo. Tra i numerosi settori in cui vengono impiegati strumenti di simulazione, si possono trovare ad esempio applicazioni in campo scientifico, tecnologico, ludico o nel campo commerciale l'allocazione di risorse e la pianificazione aziendale.

I vantaggi nell'uso della simulazione nel processo di progettazione sono numerosi. La simulazione viene ampiamente utilizzata per ridurre i rischi legati alla creazione di nuovi sistemi o alla modifica di quelli esistenti, per verificare se gli investimenti saranno in grado di produrre i risultati desiderati. Nell'industria automobilistica ad esempio per testare in fase preliminare le caratteristiche di nuovi veicoli vengono realizzati dei modelli virtuali che permettono di eseguire verifiche direttamente in un am-

biente simulato, ottenendo un grande risparmio economico rispetto all'esecuzione degli stessi test su prototipi appositamente realizzati. La simulazione al computer garantisce l'ulteriore vantaggio di avere un maggiore grado di controllo sui risultati e sugli scenari analizzati, potendo valutare con precisione le forze in gioco su ogni sezione del prototipo virtuale.

Dai prototipi alla simulazione computerizzata

Prima dell'avvento del calcolatore e dell'informatica, la realizzazione di modelli tramite replica, in scala o a grandezza reale, di un determinato scenario, per valutare, visualizzare gli effetti di un'idea, strategia o progetto e testarne il funzionamento, rappresentava un tipico esempio di simulazione.

Già prima dell'avvento dei calcolatori era comune utilizzare tecniche di modellazione risolte manualmente con formule algebriche per studiare sistemi non troppo complessi, ma è appunto grazie alla nascita e all'adozione del computer, con la costante crescita della potenza di calcolo, che è stato possibile raggiungere risultati precedentemente impensabili non solo in termini di dimensione e complessità dei problemi analizzati ma anche di rapidità nella modellazione e modalità di visualizzazione dei risultati.

Inizialmente i risultati delle simulazioni venivano rappresentati tramite tabelle o matrici che permettevano di mostrare gli effetti delle modifiche ai parametri della simulazione, in maniera analoga alle matrici che tradizionalmente venivano adottate nella modellazione matematica. Il crescente sviluppo delle capacità grafiche dei computer ha permesso la transizione verso strumenti per la rappresentazione grafica dei risultati più immediata ed intuitiva sino ad arrivare alla visualizzazione tridimensionale in tempo reale di interi scenari simulati.

Vantaggi e svantaggi dell'uso dei simulatori nella robotica

La robotica è un campo in cui l'adozione di un software di simulazione può portare a grandi vantaggi pratici. L'elevato costo dell'hardware spesso costituisce una barriera per la diffusione della ricerca e dei progressi.

Inoltre i programmi dei robot sono in genere molto complessi e come tutti i programmi di una certa dimensione richiedono tipicamente di essere ben collaudati in quanto difficilmente esenti da difetti. La loro programmazione sarà dunque composta da continue fasi di programmazione e di verifica del software sul campo. Testare ogni modifica sul robot reale può richiedere una notevole quantità di tempo e la disponibilità, durante tutta la fase di sviluppo, dell'hardware su cui si vuole lavorare. L'iter

di sviluppo può essere notevolmente velocizzato lavorando al codice e testandolo in fase preliminare direttamente su un simulatore. Una volta terminato lo sviluppo ed effettuati gli opportuni test con relative correzioni, sarà possibile spostarsi alla fase di verifica sul campo.

L'utilizzo della simulazione, nel campo della robotica può essere utile sotto diversi aspetti:

- dal punto di vista economico l'esecuzione di un esperimento tramite simulazione praticamente non ha costi, dove invece una prova sul campo potrebbe richiedere l'impiego di risorse non trascurabili;
- la simulazione permette lo studio di scenari possibili, non ancora realizzabili, in attesa di realizzazione o non facilmente accessibili;
- nel caso di non ripetibilità di un esperimento, una simulazione può fornire indicazioni preliminari su come agire poi nella situazione reale;
- la simulazione consente di avere un maggiore controllo sull'esperimento, permettendo di regolare il tempo di esecuzione, i vari parametri in gioco, ispezionare in ogni momento tutte le variabili, visualizzare l'evoluzione dell'esperimento;
- l'uso di un simulatore permette di velocizzare i tempi di sviluppo dei programmi per i robot, permettendo di testare rapidamente e senza necessità di spostarsi sul campo reale per avere un primo riscontro;
- la simulazione non richiede la presenza dell'hardware (i robot) per eseguire esperimenti. Questo può risultare molto utile ad esempio nel campo didattico, dove può risultare difficile dotare ogni studente di un robot su cui lavorare.

Come sempre, a fianco dei numerosi vantaggi permessi dall'uso della simulazione vanno considerate anche le problematiche che necessariamente verranno introdotte.

Un aspetto che accomuna le simulazioni effettuate al computer è l'approssimazione introdotta dalla limitata capacità di calcolo e dalle tecniche di risoluzione impiegate. Ci sarà sempre un compromesso tra fedeltà alla situazione originale e potenza di calcolo richiesta, limitando in precisione e accuratezza i risultati della simulazione. I risultati vanno dunque analizzati e studiati tenendo in considerazione le limitazioni introdotte dallo strumento software utilizzato.

Inoltre sebbene la fase di sviluppo direttamente in ambiente simulato permetta un notevole risparmio di tempo, sarà comunque necessaria una fase di trasferimento sul campo reale. Il robot virtuale, seppur progettato per rappresentarlo con la maggiore fedeltà possibile, non sarà mai perfettamente coincidente con quello originale.

Classificazione dei software di simulazione

Esistono diversi tipi di simulatore, tutti accomunati dall'obiettivo di fornire uno scenario rappresentativo per un modello in cui una enumerazione completa di tutti gli stati possibili sarebbe proibitiva o impossibile.

I modelli utilizzati per la simulazione possono essere classificati a seconda di diverse caratteristiche, tra le quali si possono distinguere:

- in base a come vengono modellati gli input possibili, stocastici se le possibilità o eventi casuali vengono modellati utilizzando generatori di numeri casuali o deterministici altrimenti;
- in base a come viene elaborata la risposta agli ingressi, dinamici o stazionari;
- in base a come viene rappresentato l'avanzare del tempo nella simulazione, continui o discreti;
- in base a dove vengono eseguite le simulazioni, distribuite tra più nodi interconnessi o eseguite localmente.

L'evoluzione dei calcolatori con il continuo aumentare delle capacità di calcolo e di visualizzazione dei risultati ha permesso di sviluppare simulazioni sempre più realistiche e accessibili, fino ad arrivare alla simulazione in tempo reale con la visualizzazione tridimensionale di interi mondi, non solo per scopi economici o produttivi, ma addirittura a scopi ludici oltre che educativi.

Il panorama attuale degli strumenti di simulazione è talmente vasto e frammentato che sarebbe difficile fornire una adeguata rappresentazione senza definire un qualche schema di classificazione che tenga conto delle caratteristiche e degli obiettivi degli strumenti analizzati.

Un primo aspetto che accomuna qualsiasi genere di simulazione e che in sostanza ne rappresenta il limite principale, è che una simulazione non sarà mai in grado di rappresentare esattamente la realtà, ma introdurrà un certo *livello di approssimazione* di cui si dovrà tenere conto nell'analisi dei risultati e nella formulazione di conclusioni. Il grado di approssimazione dei vari aspetti rappresentati dipenderà dagli obiettivi per cui la simulazione viene predisposta e costituisce un aspetto fondamentale nella fase di progettazione e realizzazione. Il limite superiore alla qualità della simulazione è in primo luogo rappresentato dalla limitata capacità di calcolo a disposizione, a cui vanno aggiunte considerazioni sugli strumenti e sulle tecniche utilizzate per riprodurre lo scenario previsto. Sarà dunque essenziale stabilire a priori per ogni aspetto simulato il grado di fedeltà che si intende ottenere, tenendo sempre in considerazione il compromesso tra accuratezza e sostenibilità a livello computazionale della simulazione.

Si potranno inoltre distinguere simulatori sviluppati *ad hoc* per rappresentare con grande accuratezza e livello di dettaglio le caratteristiche di interesse di uno scenario predefinito, in contrapposizione ad altri strumenti di uso generico, o *general purpose*, progettati per un utilizzo più generale ma non predefinito, che dovranno di conseguenza fornire un grado di accuratezza più limitato ma equamente distribuito fra tutte le componenti della simulazione.

Un'altro grado di classificazione dei programmi di simulazione può essere basato sull'*ambito di utilizzo* a cui è destinato un simulatore. È comune ad esempio trovare simulatori utilizzati in ambito didattico, per addestramento, per esperimenti non ripetibili, competizioni, verifica di ipotesi, test di prodotto o in campo ludico.

In base al livello di precisione e dunque alla quantità di calcoli e dalle tecniche previste per l'elaborazione, sarà possibile stabilire una ulteriore distinzione tra i software di simulazione *online*, in grado di eseguire l'avanzamento della simulazione in tempo reale di pari passo con la visualizzazione e interagendo con l'utente, e simulatori *offline* che eseguono in fase preliminare tutti i calcoli necessari per consentire solo successivamente la visualizzazione dei risultati.

Un ulteriore elemento di classificazione, più che sulle caratteristiche presentate basato sulle modalità di utilizzo e sulla possibilità di adattamento e modifica, riguarda il tipo di licenza con cui è distribuito il software. Mentre in generale un prodotto con licenza di tipo *proprietario* può presentare vantaggi in termini di completezza e disponibilità dei servizi di supporto, un progetto *open source* può garantire una maggiore libertà di utilizzo e con la disponibilità del codice sorgente permette in caso di necessità la modifica o l'estensione del software secondo le proprie esigenze.

Ulteriori criteri di classificazione si possono determinare sulle *modalità di visualizzazione* dei risultati, sulle modalità di *interazione con i robot* simulati, sul fatto che la simulazione venga eseguita su un unico calcolatore o *distribuita* tra più nodi.

Riassumendo le considerazioni effettuate è possibile delineare una serie di caratteristiche che consentono di catalogare i numerosi sistemi di simulazione disponibili per l'uso nel campo della robotica:

- ambiente di simulazione ad-hoc (situazione particolare) o simulatore general purpose;
- modalità di presentazione dei risultati: grafica (2D o 3D) o di altro tipo;
- modalità di controllo dei robot;

· simulazione centralizzata o simulazione distribuita;
· simulazione online o offline;
· livello di dettaglio della simulazione;
· licenza: open source o proprietaria (commerciale o meno).

1.2 Il progetto VLAB

L'oggetto di questa tesi consiste nello sviluppo di un ambiente virtuale per la simulazione di esperimenti robotici condivisi attraverso la rete. Il lavoro riprende gli obiettivi realizzati dal progetto VLAB [2] inquadrandoli nel panorama dei software attualmente a disposizione, aggiornando i requisiti sulla base delle caratteristiche tipicamente richieste ai programmi di questa categoria.

Il progetto VLAB, nato dalla collaborazione tra il Dipartimento di Elettronica ed Informatica dell'Università di Padova, il LADSEB-CNR e l'ENEA, aveva come scopo la realizzazione di un laboratorio virtuale al quale più utenti potessero accedere attraverso la rete. Il simulatore era strutturato per consentire la creazione di ambienti sperimentali diversi per ambito di studio, precisione della simulazione, interazione tra entità in gioco e caratteristiche ambientali, con possibilità di definire le caratteristiche dei sensori e degli attuatori dei robot considerati.

Gli obiettivi e le caratteristiche delineati dal progetto VLAB sono i seguenti:

- *Condivisibilità via rete:* l'ambiente doveva essere accessibile attraverso la rete Internet da una pluralità di utenti potenzialmente dislocati ovunque.

- *Accesso concorrente:* l'accesso al laboratorio virtuale poteva essere effettuato da un numero variabile di utenti che partecipano (attivamente con l'inserimento di robot o anche solo passivamente per visualizzare i risultati) alla simulazione.

- *Architettura client-server:* il server, una macchina dotata di grande capacità di calcolo, gestisce la simulazione, mentre i client forniscono un'interfaccia per visualizzarne i risultati e fornire un controllo sull'esecuzione.

- *Sensorialità e capacità attuative degli agenti:* possibilità di rappresentare le caratteristiche dei sensori e degli attuatori in fase di inserimento nella simulazione.

- *Simulazione multimediale del laboratorio:* disponibilità di interfacce particolarmente ricche di informazioni per facilitare la valutazione dei risultati ottenuti dalle simulazioni

- *Estendibilità:* possibilità di creazione di nuovi ambienti o aggiornamento di quelli inizialmente a disposizione.

1.3 Lo stato dell'arte

Gli obiettivi ereditati dal progetto VLAB sono stati inquadrati nel panorama dell'attuale disposizione di software per la simulazione nel campo della robotica. A questo scopo è stata effettuata una selezione rappresentativa dei programmi di simulazione attualmente a disposizione. Per ogni prodotto considerato è stata svolta una valutazione delle caratteristiche presentate, in modo da comporre una analisi dello stato dell'arte sulla quale basare la revisione delle caratteristiche da realizzare per il simulatore.

La selezione dei progetti da analizzare è stata mirata alla creazione di un gruppo il più possibile eterogeneo, cercando di comprendere tutta la gamma delle caratteristiche delineate nelle sezioni precedenti per la classificazione di questo tipo di software. Tra i progetti selezionati possiamo trovare pacchetti proprietari, commerciali o gratuiti, oltre a numerosi software di tipo open source, spesso preferiti per la maggiore flessibilità delle licenze e possibilità di estensione in base alle esigenze degli utilizzatori. Si è passato da progetti di uso generico e comprensivi di un gran numero di strumenti integrati, a software ad hoc realizzati per la simulazione di specifici contesti, limitati per dimensioni o ambito di utilizzo.

Di seguito viene riepilogata una sintesi delle caratteristiche evidenziate per ogni progetto studiato.

1.3.1 Progetti con licenza open source

The Player project

Il progetto Player [22], inizialmente sviluppato al laboratorio di ricerche sulla robotica della University of Southern California, ma attualmente sviluppato da un team di ricercatori a livello internazionale. Il progetto è costituito attorno a tre componenti principali.

Player è un server in grado di fornire un'interfaccia per consentire, una volta installato su robot fisici, il controllo dei robot ed i loro sensori attraverso la rete.

Stage, un simulatore di robot veloce e scalabile, in grado di simulare una popolazione di robot mobili, sensori ed oggetti in un ambiente bidimensionale. Utilizza l'interfaccia Player per l'accesso ai robot simulati attraverso la rete.

Gazebo [23] un simulatore multirobot, inizialmente sviluppato per rappresentare ambienti esterni ma utilizzabile anche per ambienti interni, in grado di simulare robot, sensori ed oggetti in un mondo tridimensionale. Include un motore di simulazione per la fisica dei corpi rigidi per fornire una feedback realistico e fisicamente plausibile ai sensori simulati. Anche Gazebo consente l'accesso ai robot simulati tramite l'interfaccia player in aggiunta alla propria interfaccia nativa.

Caratteristiche
Ambito di utilizzo: general purpose.
Modalità di visualizzazione: 2D Stage, 3D Gazebo.
Accesso ai robot: basata sulla rete.
Architettura: simulazione locale, realtime.
Licenza: GPL.
Note: modelli dei robot e degli oggetti simulati realizzati come plugin.

RoboCup Soccer Simulator/

Il RoboCup Soccer Simulator [25] (rcssserver) è uno strumento utilizzato per la ricerca e per scopi didattici nel campo dei sistemi multiagente e dell'intelligenza artificiale. Consente la simulazione di competizioni tra squadre di undici robot per la Robocup Soccer League in ambiente bidimensionale. È costituito da un server al quale i client inviano attraverso la rete i comandi che i robot devono eseguire. Il server non ha una propria interfaccia di visualizzazione dei risultati della simulazione ma è disponibile uno strumento chiamato monitor sviluppato a questo scopo. Il RoboCup Simulated Soccer Server 3D (rcssserver3d) in maniera analoga è in grado di simulare una partita di calcio tra robot in ambienti tridimensionali ed è alla base della RoboCup 3D Soccer Simulation League.

Caratteristiche
Ambito di utilizzo: ad-hoc, simulazione per competizioni RoboCup Soccer.
Modalità di visualizzazione: strumenti monitor 2D o 3D.
Accesso ai robot: invio dei comandi attraverso la rete.
Architettura: client-server, realtime.

Licenza: GPL.

USARSim

USARSim[24] è un simulatore di robot ed ambienti tridimensionali ad alta fedeltà di rappresentazione, inizialmente progettato per l'uso in ambito di Urban Search And Rescue (USAR) ma attualmente utilizzato in numerosi campi di ricerca ed impiegato inoltre per le competizioni RoboCup Rescue Virtual Robots e per l'IEEE Virtual Manufacturing Automation Competition. Lo strumento è basato sul motore del videgioco Unreal Turnament che consente una buona fedeltà nella rappresentazione degli scenari e nella gestione della fisica. Sono disponibili una serie di modelli predisposti per ambienti, robot e sensori da utilizzare nelle simulazioni.

Caratteristiche
Ambito di utilizzo: general purpose.
Modalità di visualizzazione: 3D.
Accesso ai robot: supporto di diverse interfacce per i robot.
Architettura: simulazione locale, realtime.
Licenza: GPL.

Simbad

Simbad[26] è un simulatore di robot 3D sviluppato in java per scopi scientifici ed educativi, dedicato principalmente alla ricerca nei campi dell'Intelligenza Artificiale, del Machine Learning e più in generale per lo studio di algoritmi nel contesto di robot e agenti autonomi. La simulazione è mantenuta volontariamente semplice, non intende fornire una simulazione realistica della realtà. Simbad permette la realizzazione dei programmi per i robot utilizzando Java o scrivendo script in Python.

Caratteristiche
Ambito di utilizzo: general purpose.
Modalità di visualizzazione: visualizzazione.
Accesso ai robot: API Java, script Python.
Architettura: simulazione locale, realtime.
Licenza: GPL.

breve

Breve[27] è un ambiente progettato per la simulazione multi agente, in grado di rappresentare agenti autonomi in un ambiente tridimensionale, includendo il supporto per un motore per una gestione realistica della fisica, un linguaggio di scripting per la definizione delle simulazioni in grado di gestire automaticamente le comunicazioni, la rappresentazione nello spazio 3D e la visualizzazione grafica. È possibile interfacciarsi con il sistema simulato tramite l'utilizzo di un'opportuna interfaccia di programmazione basata sui plugin.
Caratteristiche
Ambito di utilizzo: general purpose.
Modalità di visualizzazione: 3D.
Accesso ai robot: API plugin.
Architettura: simulazione locale.
Licenza: GPL.

Rossum's Playhouse (RP1)

Rossum's Playhouse [28] è un semplice simulatore in ambiente bidimensionale per lo sviluppo degli algoritmi per la logica di controllo e navigazione dei robot. Il software è principalmente uno strumento per aiutare lo sviluppo di applicazioni per la robotica e può essere utilizzato per la verifica degli algoritmi e la logica di controllo. RP1 supporta sia applicazioni per la risoluzione di labirinti che situazioni dove l'ambiente sia noto a priori. La programmazione dei robot può essere effettuata utilizzando l'API nativa in Java o tramite un'interfaccia di programmazione in C++.

Caratteristiche
Ambito di utilizzo: general purpose.
Modalità di visualizzazione: 2D.
Accesso ai robot: API Java e C++.
Architettura: client-server, realtime.
Licenza: GPL.

1.3.2 Software con licenza proprietaria

Webots

Webots[29] è un ambiente di sviluppo utilizzato per modellare, programmare e simulare robot mobili in un ambiente condiviso. Il sistema è accompagnato da una buona quantità di documentazione ed è disponibile un servizio di supporto utente. Tra le varie caratteristiche principali possiamo notare l'utilizzo di un motore di gestione realistica della fisica basato su ODE, il formato di file webots per la rappresentazione dei robot, API per la programmazione in sei diversi linguaggi di programmazione, simulazione realistica della telecamera, visualizzazione grafica 3D dei risultati della simulazione, un editor integrato. Sono inoltre disponibili librerie di robot predefiniti e oggetti premodellati, oltre a numerosi esempi di programmazione di robot.

Caratteristiche
Ambito di utilizzo: general purpose.
Modalità di visualizzazione: 3D.
Accesso ai robot: API in sei linguaggi di programmazione.
Architettura: simulazione locale, realtime.
Licenza: commerciale.

anyKode Marilou

AnyKode Marilou[30] è un ambiente di modellazione e simulazione progettato per facilitare l'accesso a progetti di sviluppo legati alla robotica. L'obiettivo del progetto è la realizzazione di un motore in grado di rappresentare con un alto livello di fedeltà il comportamento di sensori e attuatori. L'ambiente è dotato di uno strumento grafico per la gestione dei modelli dei robot e degli ambienti, di librerie di sensori, di robot (esistenti e viruali), di ambienti di simulazione ed è dotato di un motore per la simulazione della fisica e dei suoni. La simulazione di sistemi multi robot con visualizzazione tridimensionale può essere eseguita in tempo reale o accelerata, con la possibilità di applicazioni centralizzate o distribuite tra simulazioni eseguite su diversi computer. La programmazione dei robot avviene attraverso una interfaccia C++ e per linguaggi .NET che consente l'accesso ai robot anche attraverso la rete.

Caratteristiche
Ambito di utilizzo: general purpose.

Modalità di visualizzazione: 3D.
Accesso ai robot: API C++ e .NET.
Architettura: simulazione centralizzata o distribuita, realtime o accelerata.
Licenza: commerciale.

EyeSim - The EyeBot simulator

EyeSim[31] è un simulatore di robot Eyebot che consente la sperimentazione con gli stessi programmi che vengono eseguiti sui robot reali. Eyesim comprende la simulazione di tutti i sensori e attuatori del robot Eyesim, compresi sensori infrarossi, i bumper, sensori odometrici e visione. L'interfaccia utente permette di visualizzare una rappresentazione in 3D della scena simulata oltre a consentire l'acceso a tutti i robot. L'accesso da parte dei programmi dei robot all'ambiente simulato è consentito tramite l'uso della stessa interfaccia utilizzata per la programmazione del robot reale, consentendo dunque di utilizzare lo stesso codice per la simulazione e per il funzionamento sul campo.

Caratteristiche
Ambito di utilizzo: ad-hoc.
Modalità di visualizzazione: 3D.
Accesso ai robot: la stessa API con cui viene programmato il robot.
Architettura: simulazione locale, realtime.
Licenza: freeware.

Microsoft Robotics Studio Visual Simulation Environment

Microsoft Robotics Developer Studio (MRDS)[32, 33] è un ambiente concepito per la creazione di applicazioni nel campo della robotica pensato per ricercatori, hobbisti e sviluppatori in ambito commerciale. MRDS comprende numerosi strumenti, tra i quali il Visual Simulation Environment (VSE).

Il simulatore è composto da diversi componenti e costruito con un architettura basata sui servizi, da quelli per la gestione della fisica dei corpi (consentendo se disponibile di utilizzare l'accelerazione hardware attraverso l'uso della tecnologia NVIDIA PhysX), al servizio per la visualizzazione tridimensionale dell'avanzamento della simulazione. L'inserimento degli elementi della simulazione sotto forma di componenti software (Entities) che si interfacciano con il motore della fisica e con quello di visualiz-

zazione. L'accesso alle entità simulate è consentito tramite la realizzazione di appositi servizi (Services).

La creazione dell'ambiente simulato può avvenire direttamente tramite codice di programmazione da compilare assieme alle librerie del simulatore o tramite di descrizione in formato xml per l'utilizzo di uno dei cinque scenari disponibili (Apartment, Factory, Modern House, Outdoor, Urban).

Caratteristiche
Ambito di utilizzo: general purpose.
Modalità di visualizzazione: 3D.
Accesso ai robot: servizi da realizzare per ogni entità rappresentata.
Architettura: simulazione locale, orientata ai servizi, realtime.
Licenza: commerciale.

V-REP - the Virtual Robot Experimentation Platform

V-REP (Virtual Robot Experimentation Platform) [34] è un simulatore di robot tridimensionale dotato di un ambiente integrato di sviluppo. I programmi per i robot possono accedere alla simulazione con diversi approcci, tra i quali una API di programmazione C++, una API remota per l'accesso tramite la rete, l'utilizzo di script in linguaggio Lua o tramite il supporto all'interfaccia ROS. V-REP permette di modellare e simulare interi sistemi robotici o singoli sottosistemi. Comprende due differenti motori (Bullet e ODE) per la gestione della fisica e della collisione tra gli oggetti e supporta la rappresentazione di particelle configurabili che possono essere utilizzate per simulare getti d'aria o d'acqua, motori jet o propellenti. In dotazione una libreria estensibile di modelli utilizzabili tramite un opportuna interfaccia visuale di gestione e composizione delle scene. Permette inoltre la simulazione del taglio di superfici utilizzando diversi strumenti di taglio configurabili.

Caratteristiche
Ambito di utilizzo: general purpose.
Modalità di visualizzazione: 3D.
Accesso ai robot: API C/C++, API remota basata sulla rete, script, interfaccia ROS.
Architettura: simulazione locale, realtime.
Licenza: commerciale.

Capitolo 2

Progettazione del sistema

2.1 Analisi dei requisiti

Gli obiettivi di questo progetto sono la progettazione e lo sviluppo di un ambiente software in grado di gestire la simulazione di laboratori virtuali utilizzati per eseguire esperimenti nel campo della robotica. Tale sistema potrà essere utilizzato ad esempio per scopi didattici, per verificare ipotesi di sviluppo o per testare i programmi per il comando dei robot permettendo di ridurre i tempi di realizzazione.

I requisiti principali comprendono la possibilità di condivisione della simulazione tramite la rete, la possibilità di rappresentare le caratteristiche di azione e percezione dei robot simulati e la capacità di fornire una visualizzazione tridimensionale dei risultati della simulazione.

A seguito di una analisi sullo stato dell'arte nel campo della simulazione per la robotica, i requisiti di base ereditati dal progetto VLAB sono stati affiancati da una serie di caratteristiche ritenute necessarie per la realizzazione di uno strumento in grado di soddisfare pienamente le esigenze dei potenziali utilizzatori.

Alcune delle caratteristiche tipicamente sfruttate nell'uso di strumenti di simulazione e che costituiranno obiettivi di base da realizzare per il progetto sono ad esempio la possibilità di eseguire un *controllo sull'esecuzione* della simulazione, effettuare istantaneamente una *valutazione dei valori percepiti* dalla strumentazione e la *ripetibilità degli esperimenti senza costi aggiuntivi*.

Altro requisito fondamentale è la capacità da parte dell'utente di poter rappresentare il più liberamente possibile i propri robot e tutti gli elementi da introdurre nella simulazione. Questa caratteristica viene tuttavia controbilanciata dalla complessità e difficoltà richiesta dalla realizzazione del

modello del sistema che si intende rappresentare. Nello sviluppo di questo progetto si è cercato di realizzare uno strumento di uso generico, il più possibilmente flessibile ma in grado di *mascherare la complessità di rappresentazione del modello* tramite l'utilizzo di *elementi primitivi*, un insieme limitato di blocchi costitutivi che possono essere connessi tra loro tramite l'utilizzo di vari tipi di giunto per formare la rappresentazione di oggetti più complessi.

Diventa dunque un requisito di primaria importanza la progettazione di un opportuno *linguaggio di descrizione* che fornisca la possibilità di rappresentare composizioni più elaborate degli elementi che costituiscono gli oggetti da simulare.

Il progetto dovrà inoltre essere in grado di fornire ai programmi dei robot simulati un'interfaccia per la percezione e l'azione che consenta di interagire con il simulatore. L'utilizzo di tale interfaccia (denominata *RobotAPI*), permetterà di sostituire all'ambiente reale, in cui tipicamente agirebbe il robot, quello simulato al calcolatore.

Un altro aspetto importante è la riduzione del tempo richiesto per il passaggio dei programmi per i robot dall'ambiente simulato al campo reale. Per questo motivo viene prevista la possibilità di estendere il simulatore con il supporto all'esecuzione di codice scritto in linguaggi diversi da quello supportato nativamente. La possibilità di scrivere codice nello stesso linguaggio che verrà utilizzato sull'hardware di destinazione consente di limitare il lavoro necessario al trasferimento sul campo, richiedendo, se non già effettuata, la sola implementazione di uno strato di astrazione che permetta di agire sull'hardware utilizzando le stesse API del simulatore.

Infine ci si prefigge di realizzare del codice per rappresentare, realizzando semplici *esempi di simulazione*, le potenzialità del software realizzato.

2.1.1 Caratteristiche del sistema da realizzare

A seguito delle valutazioni sulle caratteristiche richieste al software da realizzare, è possibile sintetizzare schematicamente gli obiettivi all'oggetto di questo lavoro di tesi:
Caratteristiche ereditate da VLAB:

- condivisibilità: deve essere consentito l'accesso alla simulazione attraverso la rete
- architettura client-server: il server si occupa di eseguire la simulazione, i client ricevono i risultati dal server e ne permettono la visualizzazione

- sensorialità e capacità attuative degli agenti: deve essere possibile specificare l'insieme delle capacità di percezione ed attuazione di ogni robot

- simulazione multimediale del laboratorio: le scene prodotte dalla simulazione vengono renderizzate in tempo reale dai client

ulteriori caratteristiche da realizzare:

- possibilità di controllo dell'evoluzione della simulazione

- possibilità ispezione dei valori percepiti dagli strumenti simulati

- possibilità di introduzione di nuovi elementi in qualsiasi fase della simulazione

- capacità di eseguire software dei robot con definizione di un'interfaccia standard per l'accesso all'ambiente simulato (RobotAPI)

- possibilità di supportare l'esecuzione di codice scritto in linguaggi differenti da quello supportato nativamente

- realizzazione di una simulazione di esempio per introdurre alle caratteristiche e alle potenzialità del simulatore.

2.2 Architettura del sistema

Tra le caratteristiche che il simulatore dovrà presentare, risulta evidente l'architettura di tipo client/server del sistema.

Il server si occupa dell'esecuzione vera e propria della simulazione, ricevendo comandi e file di descrizione dagli utenti connessi e producendo i dati necessari alla visualizzazione dei risultati della simulazione.

I client permettono di controllano l'evoluzione della simulazione attraverso un'interfaccia utente, forniscono al server la descrizione dell'ambiente in fase di creazione di una nuova simulazione o possono partecipare ad una simulazione già avviata aggiungendovi nuovi elementi. I dati prodotti dal server ad ogni intervallo di simulazione vengono interpretati e ne viene fornita una rappresentazione tridimensionale attraverso l'interfaccia utente.

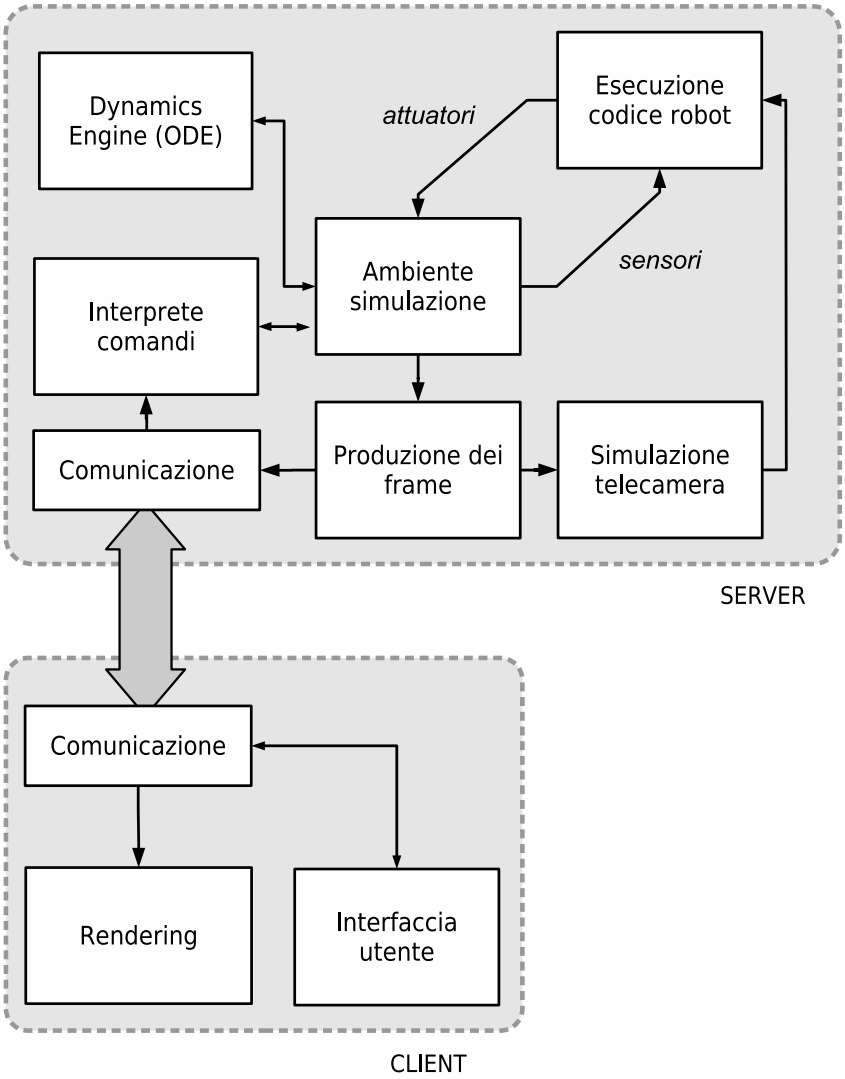

Figura 2.1: Schema generale dell'architettura del sistema

2.3 Descrizione delle componenti del sistema

L'architettura stabilita per il progetto consente di suddividerne lo sviluppo in maniera modulare, aggregando per ambito di utilizzo i diversi elementi che costituiscono il server ed il client.

2.3.1 Componenti del server

I compiti principalmente assegnati al lato server del progetto sono riassumibili in:

- Gestione dell'accesso alla simulazione e della comunicazione con i client
- Gestione e aggiornamento dello stato della simulazione
- Esecuzione dei programmi dei robot partecipanti alla simulazione
- Interpretazione ed esecuzione dei comandi e dei file descrittivi ricevuti dai client
- Produzione ed invio ai client della descrizione di ogni passo della simulazione (frame)

Comunicazione ed esecuzione dei comandi

Il *modulo di comunicazione* si occupa della gestione delle comunicazioni tra i vari client connessi e il server. Oltre alla gestione delle connessioni e l' autenticazione da parte degli utenti remoti, il modulo si occupa della ricostruzione dei comandi ricevuti e dei file di descrizione ricevuti che verranno smistati al modulo *Interprete comandi* dove verranno opportunamente valutati ed eseguiti. Il modulo di comunicazione si occupa inoltre della trasmissione ai client dei risultati prodotti ad ogni passo di simulazione.

Evoluzione della simulazione

Lo stato della simulazione, mantenuto all'interno del modulo *Ambiente di simulazione*, viene aggiornato utilizzando il *Dynamics engine*, che si occupa di applicare le forze in gioco nel sistema aggiornando, di conseguenza, le variabili associate ad ogni oggetto. Il modulo sarà dunque dedicato alla gestione della fisica del sistema garantendo l'avanzamento della simulazione tramite l' aggiornamento dello stato degli oggetti e la gestione delle collisioni.

Produzione della descrizione tridimensionale della scena

Il modulo di *produzione dei frame* si occupa di produrre la rappresentazione tridimensionale della scena, a partire dallo stato attuale della simulazione. Ad ogni istante di simulazione, viene prodotto un *frame*, una descrizione completa delle proprietà visualizzabili degli oggetti presenti nella scena. L'invio dei dati prodotti viene gestito dal modulo di comunicazione. Il modulo *simulazione telecamera* si occupa di generare, a partire dalla descrizione attuale della scena, le immagini prodotte dalle telecamere eventualmente inserite nei robot e di fornirle ai relativi programmi di controllo.

Programmi dei robot e RobotAPI

Il programma associato ad ogni robot viene inviato dal client al momento dell'inserimento nell'ambiente di simulazione. Il codice ricevuto sotto forma di sorgente viene compilato ed eseguito dinamicamente durante l'esecuzione della simulazione. L'accesso da parte dei programmi dei robot all'ambiente simulato è consentito tramite una opportuna interfaccia di programmazione chiamata RobotAPI. Oltre al linguaggio nativamente supportato dal simulatore, è previsto un meccanismo estensibile per il supporto di ulteriori linguaggi di programmazione, rendendo possibile l'esecuzione dei programmi scritti direttamente nello stesso linguaggio che verrà utilizzato sul robot fisico. Il trasferimento del programma nel campo reale potrà essere dunque limitato alla realizzazione di uno strato di astrazione dall'hardware che realizzi le interfacce del simulatore direttamente sul robot di destinazione.

2.3.2 Componenti dei client

I principali compiti attribuiti ai client sono riassumibili in:

- Gestione dei comandi ricevuti dall'utente
- Invio di messaggi di controllo al server
- Ricezione e interpretazione della descrizione dell'evoluzione della simulazione con conseguente visualizzazione a schermo

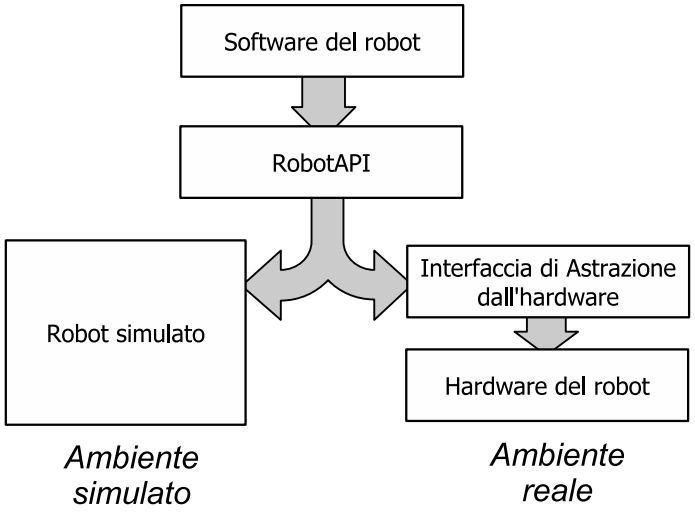

Figura 2.2: Il modulo di astrazione permette di fornire al programma di controllo dei robot un'interfaccia unificata utilizzabile sia nell'ambiente di simulazione che nel robot reale

Interfaccia utente

Il modulo *Interfaccia Utente* gestisce le operazioni di interazione con l'utente, fornendo la possibilità di controllare l'esecuzione della simulazione e impartire i comandi destinati al server.

Comunicazione

Il *modulo di comunicazione*, analogamente all'omonimo modulo del server, svolge sostanzialmente le operazioni di invio e ricezione dei dati, occupandosi di gestire la connessione con il server, inviare i messaggi contenenti i comandi impartiti dall'utente, gestire l'invio dei file descrittivi per l'inserimento di nuovi robot nella simulazione, ricevere i frame contenenti la descrizione dei risultati prodotti dal simulatore.

Creazione delle simulazioni ed inserimento dei robot

Il client fornisce all'utente la possibilità di inserire nuovi elementi nella simulazione. Per prima cosa deve essere fornita da uno dei client la descrizione dell'ambiente in cui avviene la simulazione. Successivamente ogni client può partecipare inserendo nuovi robot tramite l'invio al server dei relativi file di descrizione.

Visualizzazione della simulazione

I client permettono la visualizzazione tridimensionale dell'evoluzione della simulazione. Questa operazione viene svolta dal modulo *Rendering*, che si occupa di interpretare i dati dei frame ricevuti dal server ed eseguire il rendering della scena corrispondente.

2.4 L'esecuzione della simulazione

Una volta avviata una nuova simulazione sul server, i client che accedono al laboratorio virtuale possono inserirvi nuovi elementi, dai robot a semplici oggetti inanimati. Per prima cosa è necessario che venga fornita una descrizione dell'ambiente in cui l'esperimento viene eseguito. Successivamente i client possono agire introducendo nuovi elementi o semplicemente assistere all'evoluzione della simulazione, eseguendo il controllo sull'avanzamento, ricevere informazioni di logging dai sensori dei robot e per ogni passo di simulazione eseguito ricevono un *frame*, un'istantanea che descrive completamente lo stato degli oggetti che compongono la scena in modo da poterla rappresentare tridimensionalmente all'interno dei client. Ogni passo di simulazione è costituito da una sequenza di azioni svolte all'interno del server:

1. per ogni robot registrato viene eseguita una funzione di callback pre-esecuzione (pre-step) del plugin utilizzato per l'esecuzione del software
2. vengono applicati nella simulazione i valori assegnati agli attuatori
3. viene eseguito un passo di avanzamento dello stato della fisica del sistema simulato e successivamente vengono aggiornate tutte le strutture dati utilizzate per mantenere lo stato degli elementi simulati
4. viene prodotto un frame contenente la descrizione dello stato visivo di tutti gli elementi presenti nella scena
5. vengono aggiornati i valori percepiti dai sensori.
6. per ogni robot viene eseguito un callback post-esecuzione (post-step)

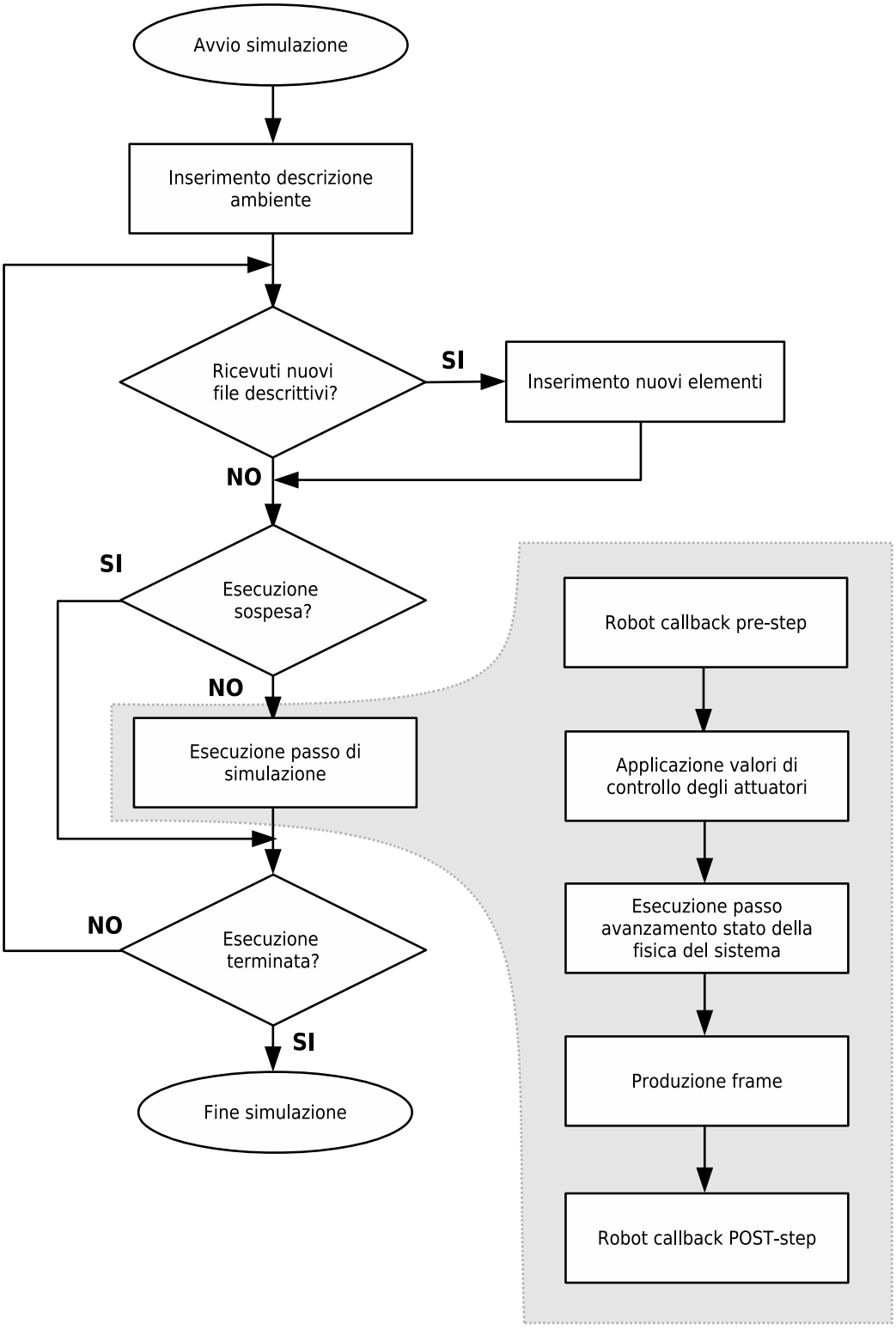

Figura 2.3: Il diagramma di flusso rappresenta i passi che compongono l'evoluzione della simulazione all'interno del server.

Capitolo 3

Realizzazione del sistema

3.1 Il motore di simulazione della dinamica

L'elemento principale del sistema, sul quale si basa interamente l'esecuzione dell'avanzamento della simulazione, è il *motore di simulazione della dinamica* (dynamics engine). Questo elemento consente di rappresentare i corpi degli oggetti presenti nella simulazione utilizzando dei modelli precostituiti e dunque evitando la necessità di specificare direttamente complesse equazioni per descrivere l'avanzamento dello stato degli elementi simulati. Oggetti più complessi possono essere ottenuti connettendo elementi primitivi tramite l'utilizzo di giunti, anche questi disponibili in una serie di classi predefinite per rappresentare in maniera semplice e immediata le possibili tipologie di vincolo tra gli oggetti interconnessi.

Oltre a gestire direttamente l'avanzamento dello stato della dinamica del sistema, calcolato utilizzando algoritmi di integrazione sulle equazioni del moto degli elementi rappresentati e delle forze ad essi applicate, il motore di simulazione utilizzato è in grado inoltre di operare una gestione delle collisioni tra gli oggetti basata sulla forma esteriore degli elementi primitivi utilizzati.

3.1.1 La libreria ODE

Data la disponibilità di numerose librerie dedicate a questo scopo, si è preferito adottare per il progetto un motore di simulazione sviluppato separatamente, consolidato e ottimizzato, anziché realizzare ex-novo un componente difficilmente avrebbe potuto garantire prestazioni migliori. Tra le alternative disponibili, sono state valutati prioritariamente progetti open-

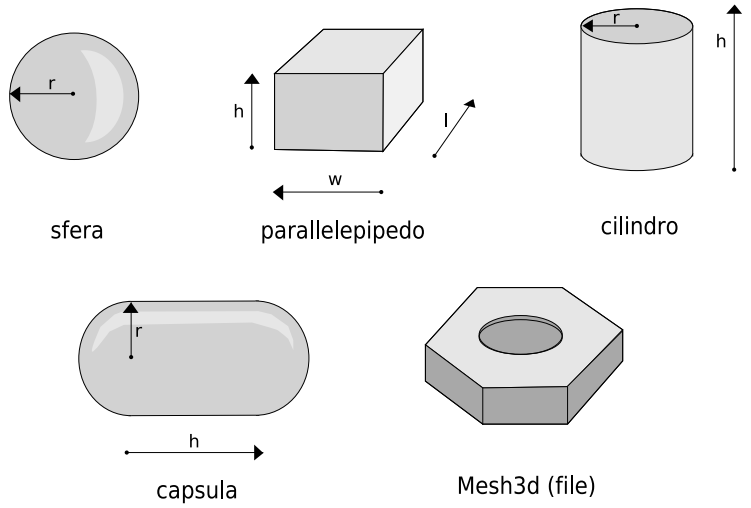

Figura 3.1: Le tipologie di elementi primitivi supportate dal motore di simulazione.

source maturi e basati su un'ampia comunità di utilizzatori, scegliendo quello più adeguato per le caratteristiche richieste dal simulatore.

Il pacchetto per la simulazione della dinamica adottato è l'Open Dynamics Engine(ODE)[20], una libreria open source di qualità industriale, utilizzata in numerose applicazioni di simulazione, da complessi pacchetti commerciali a più semplici progetti in ambito ludico, specificatamente realizzata per l'uso nelle simulazioni in tempo reale.

Inizialmente ideato e sviluppato come progetto per la tesi di laurea da Russel Smith, il progetto si è consolidato nel tempo attorno ad una comunità online di sviluppatori che tuttora ne garantisce lo sviluppo.

3.1.2 La gestione dello stato della simulazione

Il modulo di gestione dello stato della simulazione costituisce una sorta di strato di astrazione tra le entità logiche rappresentabili tramite le descrizioni fornite dagli utenti e le strutture e metodi esposti dall'interfaccia di programmazione (API) in C fornita dalla libreria ODE.

Attorno all'API del motore di simulazione utilizzato, è stata sviluppata una gerarchia di classi per consentire la gestione dello stato della simulazione e la mappatura tra gli elementi descritti dall'utente con le corrispettive strutture create all'interno del motore di simulazione.

La classe di base predisposta per la rappresentazione degli oggetti è la classe `Element`, utilizzata nei client per mantenere le informazioni stret-

tamente necessarie alla renderizzazione degli oggetti presenti nella scena (forma e dimensioni, posizione e rotazione, texture). Questa classe viene estesa, nel modulo di gestione della simulazione, dalla classe `DynamicElement` che vi aggiunge tutte le informazioni necessarie per associare l'oggetto primitivo con le strutture dati utilizzate dalla libreria per rappresentarlo, quali ad esempio la struttura *body* utilizzata per rappresentare le caratteristiche di distribuzione della massa degli oggetti e la struttura *geometry* per mantenere informazioni sulla forma esteriore, l'involucro utilizzato nella gestione delle collisioni.

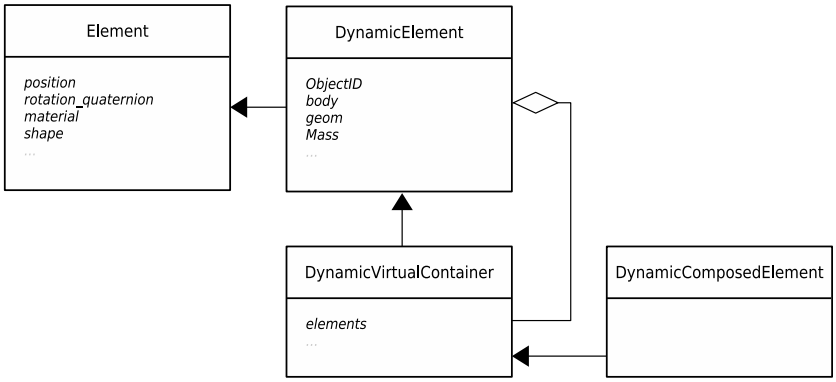

Figura 3.2: La gerarchia di classi per la rappresentazione degli elementi appartenenti alla simulazione.

A seguito di ogni passo di simulazione, il contenuto delle classi wrapper viene aggiornato utilizzando le opportune funzioni della libreria ODE. Queste informazioni saranno poi utilizzate per produrre un frame, un'istantanea che fornisce una descrizione completa di tutti gli oggetti visibili che partecipano alla simulazione e trasmessa ai client per una successiva rappresentazione.

Se da un lato la libreria di simulazione prevede l'utilizzo diretto degli elementi primitivi per rappresentare tutti gli oggetti della simulazione, dall'altro per l'utilizzatore finale è importante poter rappresentare come un'unica entità logica un insieme composto di elementi.

A questo scopo la classe `DynamicVirtualContainer` deriva dalla `DynamicElement` per raggruppare collezioni di elementi non necessariamente collegati fisicamente tra loro ma rappresentabili come un'entità unica.

La classe `DynamicComposedElement` deriva dalla classe contenitore per rappresentare elementi composti, ottenuti aggregando fisicamente tra loro più elementi primitivi, considerati nel motore di simulazione come un unico elemento monolitico.

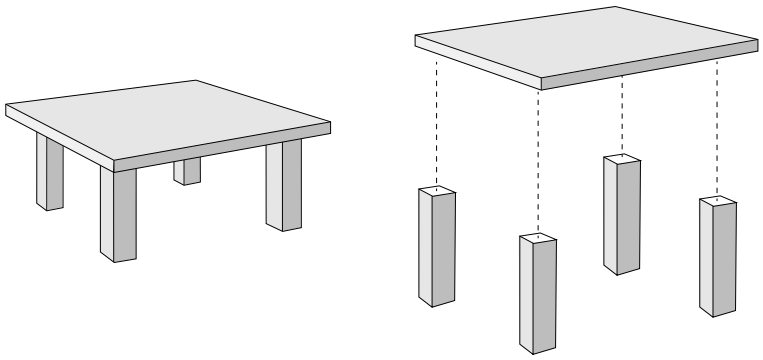

Figura 3.3: Un tipico esempio di rappresentazione di elemento composito è il caso di un tavolo, ottenuto dall'unione di una superficie piatta (base) e quattro parallelepipedi che ne compongono le gambe.

Analogamente a quanto avviene per la rappresentazione degli oggetti, la libreria ODE prevede l'utilizzo di una serie di tipi di giunto predefiniti per modellare in maniera immediata i vincoli con cui collegare fisicamente gli elementi.

Nel modulo di rappresentazione dello stato della simulazione, è stata dunque introdotta una gerarchia di classi per fornire un accesso alle caratteristiche supportate dalla libreria.

La classe astratta `Joint` rappresenta la base, contenente metodi e proprietà comuni a tutti i vari tipi supportati, sulla quale sono costruite tutte le classi wrapper che consentono di rappresentare i parametri e le proprietà caratteristiche di ogni categoria di giunto. Oltre alle caratteristiche di base, come l'indicazione degli oggetti a cui il giunto si riferisce o parametri per regolare l'elasticità nell'adesione ai vincoli, ogni classe di giunto consente di specificare una serie di proprietà specifiche, quali ad esempio l'asse (*Axis*) attorno al quale avviene la rotazione in un giunto a cerniera descritto dalla classe `JointHinge` o il punto di giunzione (*Anchor*) di un giunto a rotazione libera `JointBallAndSocket`.

3.2 Produzione e visualizzazione dei frame

Le classi realizzate per il modulo di gestione dello stato della simulazione oltre ad essere utilizzate direttamente nell'interprete del linguaggio di descrizione per mappare i contenuti descritti dall'utente svolgono un ruolo

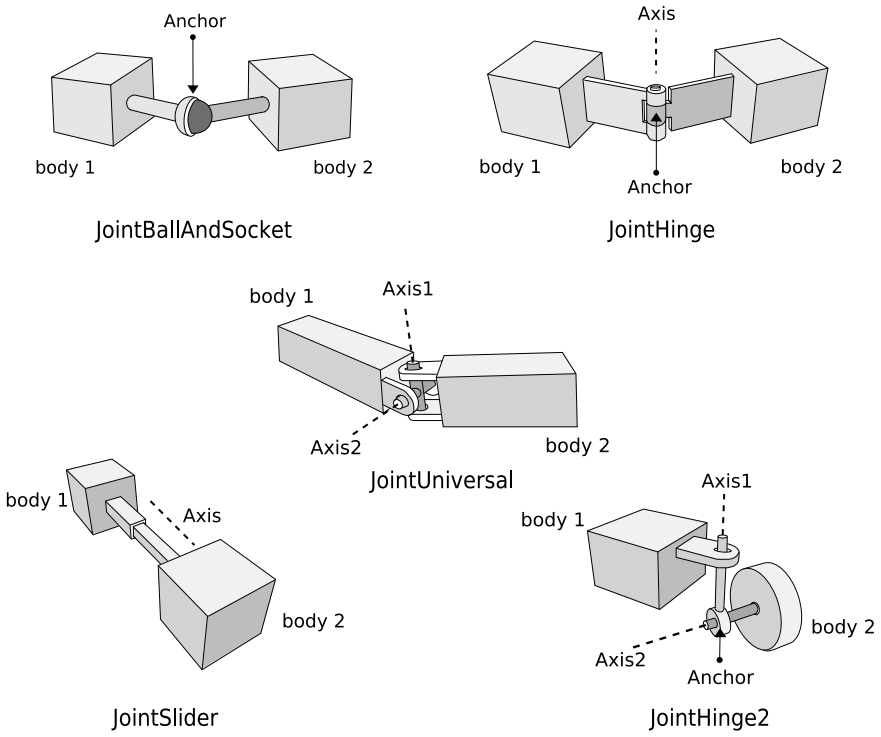

Figura 3.4: I principali modelli disponibili per i tipi di giunto.

fondamentale per la produzione dei frame utilizzati per fornire ai client una descrizione completa della scena da renderizzare.

Questi frame sono ottenuti a partire dalla lista degli oggetti partecipanti alla simulazione, un contenitore di puntatori a oggetti di tipo `DynamicElement`, serializzandone le proprietà strettamente legate allo stato visibile dell'oggetto ereditate dalla classe `Element`.

Il frame è dunque composto da una sequenza testuale che, dopo essere stata opportunamente pretrattata, viene inserita in un opportuno contenitore, un oggetto `FrameContainer`, costruito per mantenere tutti i frame prodotti e gestirne l'accesso condiviso.

I thread di gestione delle connessioni dal lato server vengono informati della disponibilità dei nuovi dati che vengono quindi inviati ai client connessi attraverso la rete. Dall'altro lato della connessione, nel client, il frame ricevuto viene inserito in un istanza locale del `FrameContainer`. Il modulo di visualizzazione riceve la notifica della presenza di nuovi frame e al momento opportuno, scandito dal timer di rendering, deserializza il contenuto del frame aggiornando un contenitore di oggetti di tipo `Element`

utilizzato per il rendering delle scene.

Per la rappresentazione tridimensionale delle scene è stata utilizzata la libreria open source Ogre (Object-Oriented Graphics Rendering Engine)[21], un motore grafico 3d dedicato in grado di mascherare, tramite un'architettura orientata alla scena, la gestione delle operazioni di renderizzazione e il supporto all'accelerazione hardware consentendo il supporto di diversi standard.

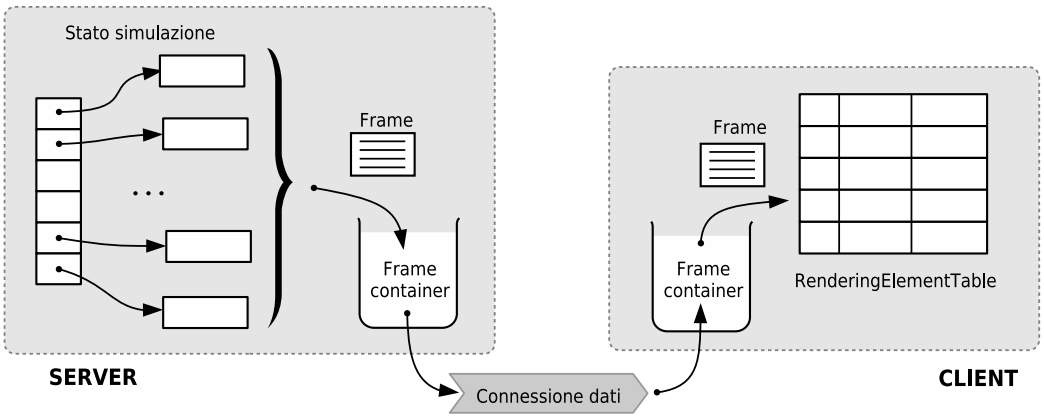

Figura 3.5: I frame prodotti nel server in seguito all'esecuzione di ogni passo di simulazioni vengono inseriti in un frame container locale, per poi essere trasmessi ad ogni client connesso, dove verranno utilizzati per costruire la tabella di rendering.

Una scena in Ogre viene descritta tramite un albero di oggetti, in cui i nodi rappresentano gli elementi da visualizzare. La gestione dell'aggiornamento delle scene è affidata alla classe `RenderingElementTable`, derivata dal semplice contenitore di elementi al quale viene aggiunta la gestione della mappatura e della sincronizzazione tra elementi e i corrispondenti nodi dello *Scene Graph*, l'albero di rappresentazione delle scene di Ogre.

Quando nella tabella di rendering viene introdotto un elemento, viene creato in corrispondenza un nuovo nodo nello Scene Graph, contenente la rappresentazione 3d opportunamente creata sulla base delle caratteristiche visive dell'oggetto. Gli elementi rappresentabili sono tutti gli oggetti primitivi gestiti dal modulo di gestione dello stato del sistema, oltre ad oggetti di tipo *Mesh3D* la cui forma esteriore, viene caricata da file in formato mesh di Ogre.

3.3 Il modulo di comunicazione

3.3.1 La gestione delle comunicazioni tra client e server

Una delle caratteristiche principali del progetto è quella di consentire un accesso distribuito alla simulazione, in modo da creare un ambiente (laboratorio) virtuale in cui i risultati delle simulazioni siano accessibili e visualizzabili da più locazioni. Questa caratteristica impone la necessità di un protocollo per regolare la gestione delle comunicazioni tra i client e il server e la realizzazione di tutta una serie di strutture necessarie per la gestione delle connessioni attive sul server. L'architettura client-server prevede dunque che vi sia una spartizione di ruoli tra il nodo principale, il server, che si occupa di produrre i risultati della simulazione e gli altri nodi, i client, che una volta stabilita una connessione verso il server potranno partecipare alla simulazione e ricevere i risultati prodotti dal nodo principale.

Per semplificare la gestione della trasmissione dei frame, la comunicazione tra ogni client e il server viene suddivisa tra due canali di comunicazione: la *connessione di controllo*, attraverso la quale i client inviano i comandi da impartire al server e ricevono i messaggi eventualmente inviati dal server, e la *connessione dati* utilizzata per l'invio ai client dei frame prodotti ad ogni passo di simulazione.

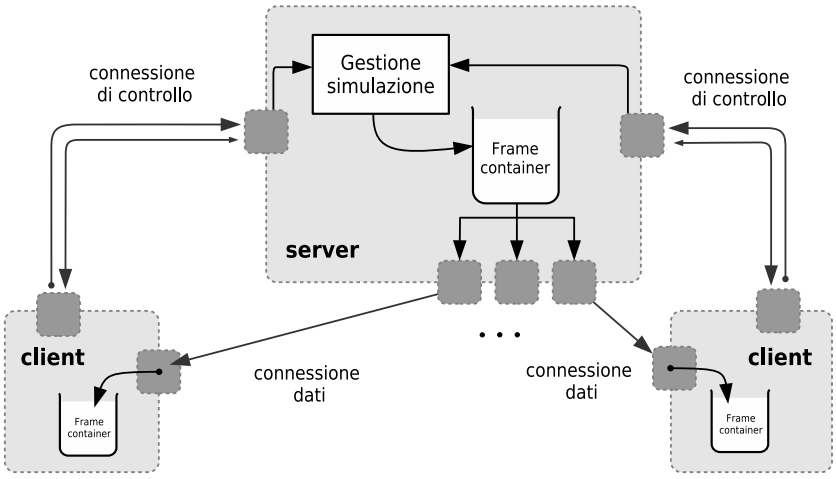

Figura 3.6: I canali di connessione stabiliti tra i client ed il server.

3.3.2 La connessione di controllo

Il canale di controllo viene stabilito dal client aprendo, in fase di accesso alla simulazione, una connessione verso la porta su cui il server rimane in ascolto.

Dopo una prima fase di riconoscimento (composta da una fase di avvio e la richiesta di accesso), il server indica al client la disponibilità all'apertura di una connessione dati. Una volta terminata con successo questa operazione, la fase di apertura della connessione viene considerata conclusa e il client, abilitato a tutti gli effetti a partecipare alla simulazione, riceverà come prima cosa tutti i frame precedentemente prodotti a partire dall'inizio dell'esperimento simulato. Da questo momento il client è in grado di inoltrare attraverso la connessione dati i comandi ricevuti tramite l'interfaccia utente, i quali una volta ricevuti dal modulo di gestione delle comunicazioni del server, dopo opportuna interpretazione e validazione, vengono applicati agendo opportunamente sul controllo della simulazione.

3.3.3 La connessione dati

Dal lato server, una volta stabilita una nuova connessione dati, il gestore della connessione si registra sul `FrameContainer` per la ricezione delle notifiche sulla disponibilità di nuovi dati. Ad ogni passo di simulazione, quando il frame prodotto viene inserito nell'opportuno contenitore, i gestori delle connessioni dati vengono risvegliati e possono dunque inviare il contenuto, opportunamente incapsulato, attraverso il canale dati. Il gestore della connessione dal lato client riceve il messaggio, estrae il frame originale e lo inserisce nel `FrameContainer` locale mettendolo a disposizione per la successiva visualizzazione.

3.4 Esecuzione dei programmi dei robot

3.4.1 La RobotAPI

L'accesso all'ambiente simulato da parte dei programmi dei robot viene consentito attraverso un'interfaccia di programmazione, la *RobotAPI*, che permette di stabilire le regole di accesso ai dati di robot, sensori e attuatori. La RobotAPI è costituita essenzialmente dai seguenti elementi:

- Gerarchia di classi astratte per fornire un'interfaccia di accesso ai valori percepiti dai sensori.

- Gerarchia di classi per l'assegnazione dei valori di comando degli attuatori.
- Interfaccia di rappresentazione dei robot (classe `Robot`).
- Interfaccia che deve essere implementata dai software dei robot (classe `RobotSoftware`) utilizzata dal simulatore per interagire con i programmi in esecuzione.

Interfacce per l'accesso ai sensori

Un sensore è un'entità che permette la rilevazione di un valore o una proprietà di un elemento della simulazione. L'interfaccia di base per l'accesso ai sensori è rappresentata dalla classe astratta `Sensor`, dalla quale derivano le interfacce per tutte le varie tipologie di sensore supportate. Queste classi, incluse nella RobotAPI, costituiscono il punto di accesso ai valori rilevati dai sensori installati nei robot, a prescindere da quali siano effettivamente le classi che effettivamente ne implementano i metodi all'interno simulatore.

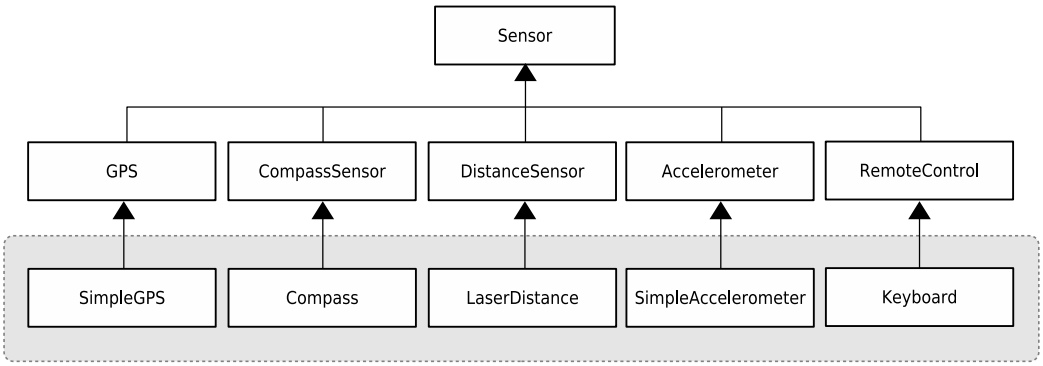

Figura 3.7: La gerarchia di sensori previsti dalla RobotAPI e le relative implementazioni all'interno del motore di simulazione.

Interfacce per l'accesso agli attuatori

Gli attuatori consentono ai software dei robot di agire sulle entità presenti nella simulazione, agendo principalmente sui giunti ai quali sono associati. Analogamente a quanto realizzato col la gerarchia di interfacce per i sensori, la classe `Actuator` costituisce la base astratta dalla quale derivano le interfacce per tutte le tipologie di attuatore previste dal simulatore.

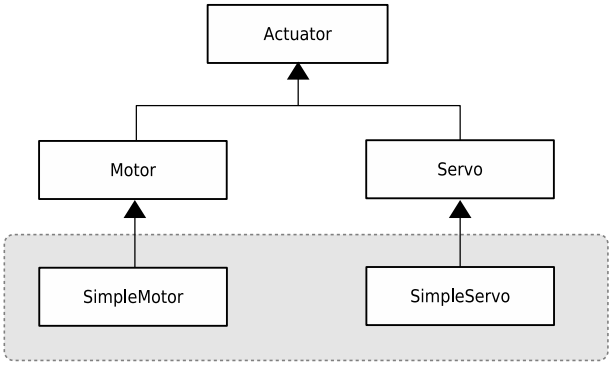

Figura 3.8: La gerarchia di attuatori previsti dalla RobotAPI e le relative implementazioni all'interno del motore di simulazione.

Interfacce per la rappresentazione dei robot ed i relativi programmi

Un robot viene rappresentato all'interno della simulazione come un gruppo di sensori e di attuatori che percepiscono e agiscono sulle proprietà di un gruppo di oggetti che costituisce il corpo del robot (*body*). La classe `Robot`, utilizzata all'interno del simulatore per mantenere le informazioni relative ai singoli robot, viene utilizzata anche nella RobotAPI come interfaccia per gestire direttamente l'accesso alle singole proprietà. I programmi dei robot, inviati dagli utenti sotto forma di sorgente in fase di inserimento di un nuovo elemento nel simulatore, vengono compilati sotto forma di plugin e caricati dinamicamente durante la simulazione. Per poter essere correttamente compilato in un plugin, il programma deve aderire alle specifiche della RobotAPI, implementando l'interfaccia `RobotSoftware` utilizzata dal simulatore per eseguire gli opportuni callback che precedono e seguono ogni passo di simulazione.

Una volta caricato, il plugin consente al simulatore di creare una istanza di un oggetto `RobotSoftware` alla quale viene assegnato un riferimento all'oggetto che rappresenta il robot comandato dal programma.

3.4.2 Compilazione dei programmi per i robot

I programmi dei robot vengono ricevuti sotto forma di file sorgente affiancati da un file progetto che descrive il modo con cui devono essere compilati. Una delle caratteristiche previste per il simulatore è la capacità di consentire l'utilizzo di programmi scritti adottando lo stesso linguaggio di programmazione utilizzato per produrre il codice eseguito direttamente sul robot reale. Per questo motivo è stato introdotto il concetto di file di progetto per specificare tutti i dettagli necessari alla compilazione dei sor-

genti forniti dall'utente e produrre un plugin. Quando riceve un progetto che deve essere compilato, il simulatore esegue la ricerca di un compilatore in grado di gestire il tipo di progetto specificato all'interno del file. I compilatori di progetto sono realizzati seguendo un'architettura a plugin, permettendo così di facilitare l'estensione del simulatore con il supporto ad ulteriori linguaggi.

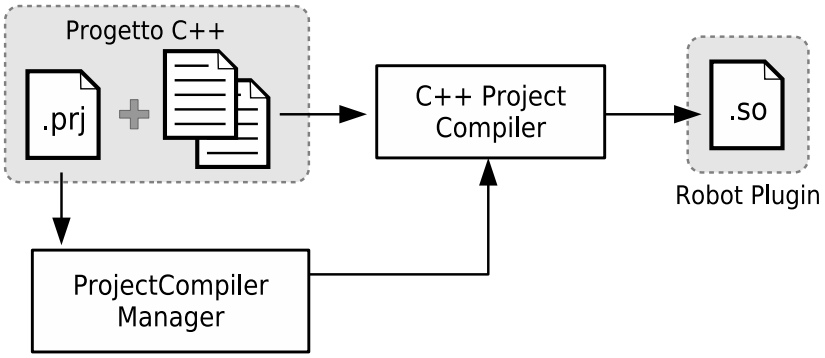

Figura 3.9: Procedura di compilazione di un progetto C++

L'aggiunta del supporto ad un nuovo linguaggio prevede dunque la realizzazione di un plugin per la compilazione dei progetti di quel tipo, oltre alla predisposizione, qualora fosse necessario, di una libreria di interfacciamento che consenta di mascherare l'accesso alle RobotAPI C++ tramite una API accessibile nel linguaggio di destinazione.

Supporto all'esecuzione di programmi Java

Oltre al linguaggio C++ supportato nativamente, è stato sviluppato il supporto ai programmi scritti in Java. A tale scopo è stata realizzata una libreria Java, chiamata *JavaRobotAPI*, che tramite l'uso della Java Native Interface (JNI) costituisce un wrapper attorno all'interfaccia nativa del simulatore.

Il relativo compilatore di progetti permette di compilare le classi ricevute sotto forma di sorgente Java e vi associa un plugin (che aderendo alla RobotAPI implementa la classe `RobotSoftware`) in grado di istanziare una Java Virtual Machine per consentire l'esecuzione del bytecode prodotto.

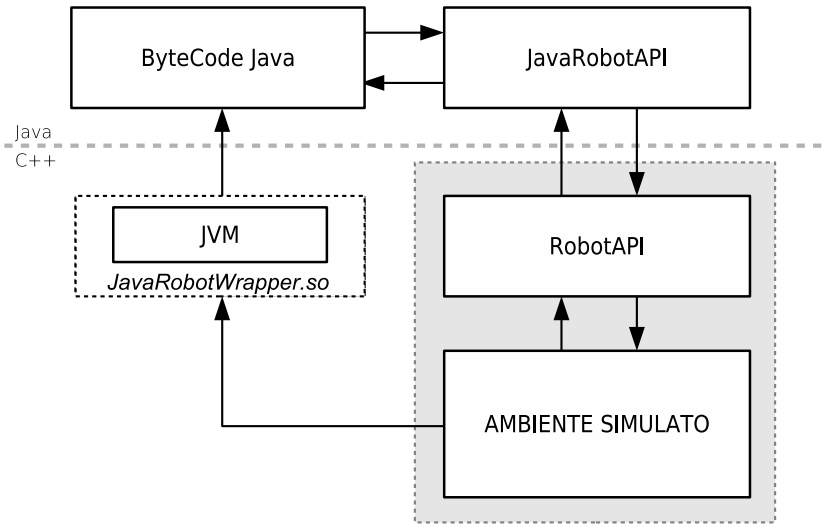

Figura 3.10: Esecuzione dei plugin per software Java e accesso al simulatore utilizzando la JavaRobotAPI

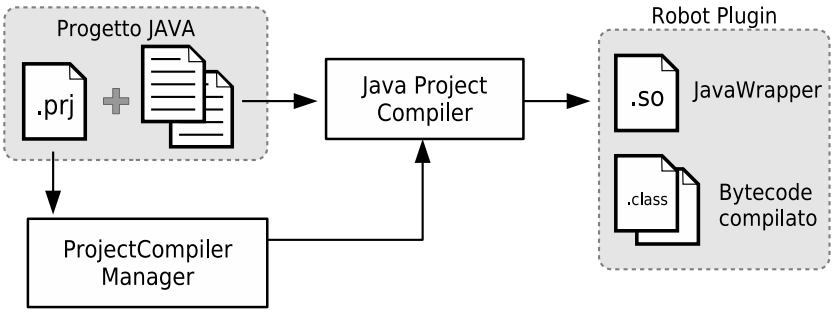

Figura 3.11: Procedura di compilazione di un progetto Java

Capitolo 4

Il linguaggio di descrizione delle scene

4.1 La descrizione delle scene

Un aspetto cruciale per delineare le capacità rappresentative del simulatore, è il metodo con cui è possibile descrivere gli elementi che partecipano alla simulazione. La descrizione di una scena sarà composta dalla descrizione dell'ambiente e di tutti gli elementi, mobili o statici, che vi prendono parte.

4.1.1 Un linguaggio specifico per il contesto

L'utilizzo di un linguaggio generico, ad esempio basato su XML, avrebbe consentito di raggiungere le capacità descrittive richieste, ma a fronte della disponibilità di parser già pronti e testati, avrebbe comportato una eccessiva complicazione dei file descrittivi che risultando di difficile comprensione e scrittura avrebbero tra l'altro richiesto lo sviluppo di uno specifico editor (che al momento esula dagli obiettivi del progetto).

Una soluzione alternativa potrebbe essere quella di prevedere che le simulazioni siano descritte direttamente tramite codice di programmazione, compilato ed eseguito assieme al simulatore, in grado di istanziare direttamente tutti gli elementi da rappresentare. Sebbene quest'ultima soluzione abbia il pregio di non richiedere ulteriori sforzi per lo sviluppo di un apposito interprete, d'altro canto costituirebbe una ulteriore complicazione per l'utente finale, riducendo la praticità complessiva del sistema negli ambiti di utilizzo previsti.

Per garantire la massima libertà espressiva e nel contempo favorire il riuso del codice sviluppato, favorendo la creazione e l'utilizzo di librerie di descrizione precostituite, si è scelto dunque di sviluppare un linguaggio specifico per il dominio considerato. La progettazione di un linguaggio e il conseguente sviluppo di uno specifico interprete, consente come vantaggio collaterale di dotare il linguaggio, con poco lavoro aggiuntivo e qualche piccola accortezza, di costrutti e funzionalità tipiche dei linguaggi di uso generico. Il linguaggio così definito non si limiterà ad una descrizione piatta degli elementi della simulazione, ma consentirà la creazione di veri e propri programmi in grado di generare la rappresentazione desiderata.

4.1.2 Requisiti per il linguaggio

Lo scopo principale del linguaggio progettato è la descrizione dell'ambiente e la rappresentazione di tutti gli elementi che partecipano alla simulazione. Il punto di partenza per la definizione del linguaggio è dunque garantire la possibilità di descrivere tutti gli oggetti rappresentabili nel simulatore con le relative proprietà. Altro fattore fondamentale è la rappresentazione delle relazioni tra i vari elementi tramite l'uso dei vari tipi di giunto con i relativi parametri, la possibilità di formare elementi composti o raggruppare più oggetti in un unica entità logica. Un ulteriore entità da rappresentare è costituita dai robot, per i quali deve essere possibile fornire una descrizione degli elementi che ne compongono il corpo, specificando tutti i parametri di sensori e attuatori, unitamente alla specifica del codice da utilizzare per il software di comando.

Oltre alla possibilità di descrizione di tutte le entità rappresentabili nella simulazione, lo sviluppo di un interprete ad-hoc ha permesso di introdurre nel linguaggio una serie di elementi e costrutti non strettamente richiesti per rappresentare le scene, ma che facilitano lo sviluppo di scenari complessi e incentivano il riuso del codice e la parametrizzazione, quali ad esempio l'uso di variabili, operatori, blocchi condizionati e cicli, la definizione e chiamata di funzioni.

4.2 Creazione di un interprete per il linguaggio

I passi richiesti per la creazione di un interprete dipendono dalle caratteristiche presentate dal linguaggio. In genere sarà comunque richiesta la presenza di un *lexer*, componente in grado di svolgere un'analisi lessicale del codice in ingresso trasformando la sequenza di caratteri in una sequenza di simboli (*token*), gli elementi costitutivi del linguaggio. Sulla sequen-

za di simboli generata dal lexer agisce un secondo componente, il *parser*, che verifica l'aderenza alla sintassi della grammatica del linguaggio. Data la natura descrittiva del linguaggio ed il tipo di esigenze evidenziate, nel caso in analisi non sono necessarie ulteriori fasi di creazione e manipolazione di rappresentazioni intermedie dell'ingresso, ma è sufficiente realizzare un interprete basato sulla sintassi (*syntax-directed interpreter*) in grado di eseguire direttamente una azione per ogni regola grammaticale del linguaggio incontrata.

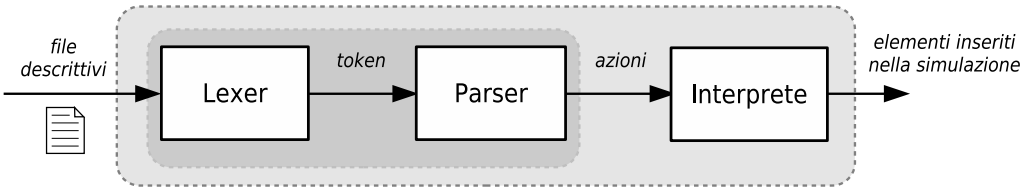

Figura 4.1: Le fasi che compongono l'interpretazione dei file di descrizione delle scene.

4.2.1 Generazione di parser e lexer con ANTLR

La scrittura di un parser per linguaggi le cui dimensioni non siano estremamente ridotte, risulta una operazione ripetitiva e soggetta ad errori, producendo codice di difficile manutenzione. Per questo motivo il settore è popolato da un grande numero di programmi in grado di gestire automaticamente la produzione del codice necessario allo sviluppo di strumenti di riconoscimento di linguaggi. La scelta dello strumento da utilizzare va ponderata principalmente in base al tipo di linguaggio utilizzato, gli algoritmi utilizzati per il riconoscimento delle regole, la praticità di utilizzo e rappresentazione della grammatica, oltre a considerazioni sulle prestazioni richieste al riconoscitore che si intende realizzare. Sulla base di questi criteri è stato dunque scelto per il progetto l'utilizzo del software open source ANTLR[19] (ANother Tool For Language Recognition), uno strumento in grado di semplificare la gestione e l'aggiornamento delle regole che compongono la grammatica del linguaggio, accompagnato da una buona quantità di documentazione.

A partire dai file che ne descrivono la grammatica, ANTLR è in grado di produrre parser a discesa ricorsiva di tipo LL(k) seguendo un approccio di tipo top-down ed è in grado dunque di lavorare su grammatiche di tipo LL(k), ovvero tutte le grammatiche libere dal contesto che non siano ricorsive a sinistra. Il supporto alla produzione di riconoscitori utilizzando

linguaggi diversi da java (con cui è stato scritto lo strumento), è garantita dalla disponibilità di numerosi backend di output. Per lo sviluppo del parser per il linguaggio di descrizione delle scene è stato appunto utilizzato il backend C, in grado di produrre codice compatibile con i compilatori C++ utilizzati per lo sviluppo del simulatore.

Il formato di descrizione della grammatica è basato sulla BNF (Backus-Naur Form) e consente opzionalmente di specificare per ogni regola, direttamente nel linguaggio di destinazione, il codice da eseguire in caso di riconoscimento.

4.2.2 Sviluppo dell'interprete

Una volta rappresentata nel formato BNF la grammatica, tramite l'utilizzo di ANTLR viene generato automaticamente il codice sorgente del lexer e del parser per il linguaggio progettato. Per completare la realizzazione del modulo di riconoscimento ed interpretazione dei file di descrizione forniti dall'utente è sufficiente realizzare un interprete in grado di eseguire azioni in corrispondenza alle regole della grammatica riconosciute dal parser.

L'interprete è costruito con una struttura che ricorda una sorta di semplice CPU, in grado di eseguire le istruzioni decodificate dal parser, mantenendo un proprio stato interno e una memoria associativa per gestire la definizione di simboli (variabili o funzioni) .

L'oggetto `SceneDescriptionInterpreter`, realizzato per questo scopo, è dotato di un metodo per l'esecuzione di ogni istruzione prevista. All'interno del parser, ogni regola riconosciuta viene fatta corrispondere ad una chiamata al metodo corrispondente nell'interprete.

Le istruzioni supportate possono essere raggruppate in categorie sulla base dell'ambito di utilizzo. Possiamo dunque distinguere istruzioni per la gestione di variabili, istruzioni per il controllo dell'esecuzione e istruzioni per la predisposizione degli oggetti da introdurre nel motore di simulazione.

La gestione dei simboli che possono essere definiti durante l'esecuzione (numeri, stringhe, valori booleani e funzioni), viene delegata alla classe `SymbolScopeStack`, una struttura con un organizzazione a pila realizzata per la gestione di contesti annidati. All'apertura di un nuovo blocco di codice, viene creato ed aggiunto nello stack un nuovo contesto nel quale verranno inserite le variabili definite all'interno del blocco. L'interrogazione per la ricerca del valore di un simbolo viene effettuata a partire dal contesto più recente, scendendo nel contesto precedente in caso di assenza del simbolo cercato, fino a raggiungere il contesto "globale" creato in

corrispondenza dell'inizio del codice. Ogni contesto viene rappresentato tramite una struttura di tipo `SymbolTable` in grado di associare il nome del simbolo all'effettivo contenuto e fornire funzioni per l'interrogazione dei simboli e la verifica della compatibilità con il tipo previsto. Nel caso in cui l'identificatore utilizzato faccia riferimento ad un simbolo di tipo non compatibile con quello atteso l'interprete segnala un errore (in questo caso di tipo semantico).

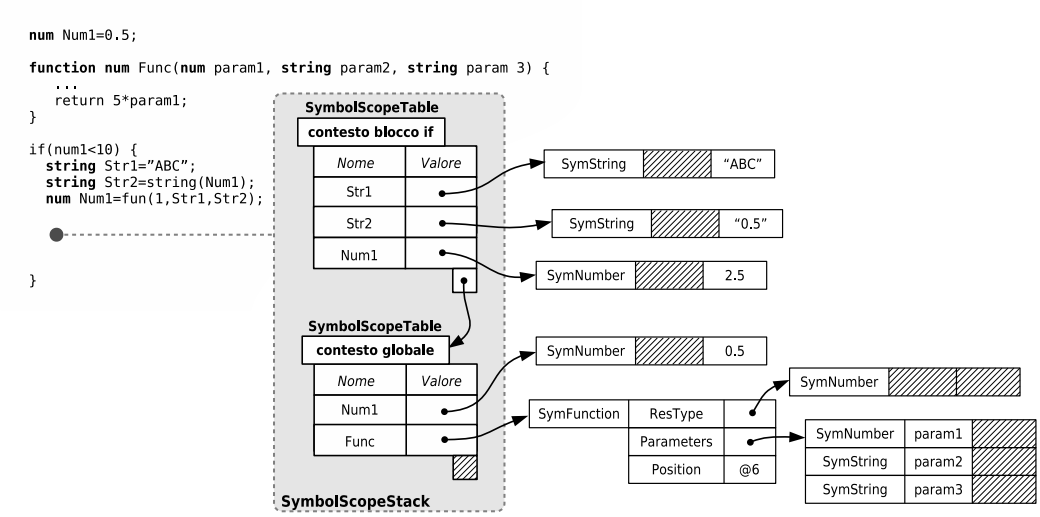

Figura 4.2: Nella figura sono rappresentate le strutture create internamente all'interprete per la gestione della definizione e l'accesso dei simboli all'interno di contesti annidati.

Le istruzioni di controllo dell'esecuzione corrispondono a possibilità di apertura e di chiusura di nuovi blocchi di codice con i relativi contesti, esecuzione condizionata ed esecuzione di cicli, definizione ed esecuzione di funzioni. Tutte queste funzionalità vengono realizzate dall'interprete interagendo direttamente con il parser, sfruttando la possibilità di memorizzare e ripristinare la posizione di esecuzione all'interno del codice di ingresso (utilizzandola come una sorta di "Instruction Pointer"), affiancata all'utilizzo di alcuni flag per la rappresentazione dello stato interno e abilitare o meno l'esecuzione delle istruzioni.

Le istruzioni dedicate alla predisposizione degli oggetti da inserire nella simulazione sono in grado di gestire la creazione e l'impostazione dei parametri di tutte le strutture dati previste dal modulo di gestione dello stato di simulazione precedentemente descritte nella sezione 3.1.2. Durante la fase di interpretazione ed esecuzione del codice descrittivo fornito dall'utente, le varie istruzioni permettono di creare ed agire su tutte queste

strutture, che vengono mantenute in uno stato disabilitato all'interno dell'interprete e inserite nel motore di simulazione una volta terminata senza errori la fase di interpretazione del codice.

4.3 Descrizione della grammatica del linguaggio

In questa sezione vengono presentate e descritte nel dettaglio le componenti del linguaggio progettato per la descrizione delle scene. Le regole della grammatica sono state suddivise sulla base dell'ambito a cui sono preposte. Per ogni gruppo, a seguito di una breve descrizione, vengono rappresentate le regole direttamente nella forma BNF utilizzata per la realizzazione del parser.

4.3.1 Elementi di base

Token elementari

Di seguito vengono descritti alcuni elementi di base del linguaggio, utilizzati nella descrizione di regole più complesse e inseriti in questo contesto come riferimento per la comprensione delle regole descritte nelle sezioni successive.

```
Space
    : (' ' | '\t' | '\r' | '\n' );

String_content
    : '"'  (~('"'))* '"'
    ;

NUMBER
    : '.'Digit+
    | Digit+('.'Digit+)?
    ;

fragment Letter:  ('a'..'z'|'A'..'Z') ;

fragment IdSymbol:  '_' ;

fragment Digit: '0'..'9' ;

fragment comparison: ('>' | '>='| '==' | '!=' | '<=' | '<' ) ;
```

Commenti

Un elemento essenziale in ogni linguaggio è la possibilità di inserire commenti, parti di codice ignorate in fase di interpretazione ma utili a livello di documentazione dei contenuti. L'inserimento dei commenti nel codice descrittivo è ispirato ai formati per i commenti previsti nei linguaggi C e C++. Il commento *in linea* viene espresso tramite la stringa "//" e comprende tutto il testo che segue fino alla fine della riga. Il commento *multilinea* consiste in tutto il testo compreso tra le stringhe di apertura "/*" e di chiusura "*/", indipendentemente dal numero di linee rappresentate.

```
LINE_COMMENT
    :   '//' ~('\n'|'\r')* '\r'? '\n' ;

COMMENT
    :  '/*' .* '*/' ;
```

Inclusione di file

Un altro elemento tipico dei linguaggi di programmazione è la possibilità di inserire codice da un file esterno, permettendo così mantenere file di dimensioni non troppo elevate e incentivando il riuso del codice. Il comando *include* carica il contenuto presente nel file specificato come parametro inserendolo nel punto in cui viene eseguita l'istruzione.

```
LINE_INCLUDE
    :
    'include'  (' ' | '\t')+ String_content  (' ' | '\t')* ';'
    ;
```

Identificatori

Un identificatore è essenzialmente il nome associato al simbolo a cui ci si intende riferire e che verrà utilizzato direttamente come etichetta nelle tabelle per la gestione dei simboli all'interno dell'interprete. Gli identificatori validi devono iniziare con una lettera e possono essere composti da lettere, numeri o il simbolo underscore ('_'). Ci sono due modi per specificare un identificatore. Il primo consiste nel specificarlo in maniera diretta inserendo i caratteri che compongono il nome.

Il metodo alternativo, utile soprattutto nel caso di generazione di un numero variabile di oggetti (ad esempio all'interno di cicli), consente di assegnare dinamicamente l'identificatore ottenendolo come risultato di operazioni fra stringhe.

```
identifier
    : (Letter|IdSymbol)(Letter|IdSymbol|Digit)* //specifica diretta
       dell'etichetta
    | 'id:''['str_expr']' //metodo alternativo che utilizza una
       espressione
    ;
```

Definizione di variabili, assegnamento ed operatori

I tipi di variabili previsti per il linguaggio sono quelli necessari per specificare i vari attributi per gli oggetti rappresentabili e per la gestione dell'esecuzione condizionata di blocchi di codice.

Le variabili numeriche (di tipo *num*) corrispondono a numeri decimali a doppia precisione, le variabili stringa (*string*) a sequenze di caratteri ed i valori booleani (*bool*) ai valori logici vero e falso.

Per ogni tipo di variabile sono previste oltre alla definizione e all'assegnazione, una serie di operatori di uso generico.

Variabili numeriche

```
num_def  //definizione di una variabile numerica (con assegnamento
    iniziale opzionale)
    :  'num' identifier ('=' num_expr)?
    ;

num_assign  //assegnamento di un valore ad una variabile numerica
    : identifier '=' num_expr
    ;

num_expr //operatori binari (a priorità inferiore)
    : num_op_prec2 ('+' num_expr )+
    | num_op_prec2 ('-' num_expr )+
    | num_op_prec2
    ;

num_op_prec2
    : num_op_prec3 ('*' num_op_prec2 )+
    | num_op_prec3 ('/' num_op_prec2 )+
    | num_op_prec3
    ;
```

```
num_op_prec3 //operatori binari (a maggiore priorità)
    : num_element ('^' num_op_prec3 )?
    ;

num_element //costanti e singole unità numeriche
    : num
    | ('pi'|'PI')
    | ('e'|'E')
    | oth_operator
    | '('num_expr')'
    | identifier
    | numeric_fun_call
    ;

oth_operator //operatori unari e altri operatori
    : rand_num
    | '-' num_element
    | 'sin''(' num_expr ')'
    | 'cos''(' num_expr ')'
    | 'sign''(' num_expr ')'
    | 'sqrt''(' num_expr ')'
    | 'log''(' num_expr ')'
    | 'log10''(' num_expr ')'
    | 'mod''(' num_expr ',' num_expr ')'
    | 'floor''(' num_expr ')'
    | 'ceil''(' num_expr ')'
    | 'round''('num_expr ')'
    ;

rand_num //numeri casuali
    : rand_key // valore casuale compreso tra 0 e 1
    | rand_key'(' n=num_expr ')' // valore casuale compreso tra 0 e n
    | rand_key'(' n1=num_expr ',' n2=num_expr ')' //tra n1 e n2
    ;

rand_key : ('random' | 'rnd') ;
```

Stringhe

```
string_def //definizione di variabile stringa (con assegnamento
    iniziale opzionale)
  : 'string' identifier ('=' str_expr )?
  ;
```

```
string_assign //assegnamento di una stringa
    : identifier ':' str_expr
    ;

str_expr //concatenazione di stringhe
    : str_expr_element ( '.' str_expr_element )*
    ;

str_expr_element
    : stringval
    | 'string''(' num_expr ')' //conversione da numero a stringa
    | identifier
    | string_fun_call
    ;
```

Valori Booleani

```
bool_def //definizione di variabile booleana
    :   'boolean' identifier ('=' bool_expr )?
    ;

bool_assign //assegnamento di un valore booleano
    : identifier ':' bool_expr
    ;

bool_expr //operatori binari (a priorità inferiore)
    : bool_op_prec_2 ( '&&' bool_expr )+
    | bool_op_prec_2
    ;

bool_op_prec_2 //operatori binari (a priorità superiore)
    : bool_op_prec_3 ( '||' bool_op_prec_2 )+
    | bool_op_prec_3
    ;

bool_op_prec_3
    : num_expr comparison num_expr  //confronto tra espressioni
        numeriche
    | bool_element
    ;

bool_element //singole entità e operatori unari
```

```
    : 'true'
    | 'false'
    | '!' bool_expr
    | '(' bool_expr ')'
    | identifier
    | 'bool''('num_expr')'
    ;
```

Utilità e rappresentazione di variabili multidimensionali

```
print_var // visualizzazione nei log del valore di una variabile
    : 'print_str' str_expr
    | 'print_num' num_expr
    | 'print_bool' bool_expr
    ;

numlist //lista di valori numerici
    : '{'( listelement(',' listelement )*)?'}' ;

listelement //elementi che possono comporre una lista
    : range
    | num_expr
    ;

range //intervallo di valori da n1 a n2 (compreso nel primo caso,
    escluso nel secondo). Se non specificato altrimenti il salto è di
    un'unità
    : n1=num_expr '>=' n2=num_expr ('|' salto=num_expr)?
    | n1=num_expr '>' n2=num_expr ('|' salto=num_expr)?
    ;

// specifica di variabili fino a quattro dimensioni
mdim_1: '(' num_expr ')';
mdim_2: '(' num_expr ',' num_expr ')';
mdim_3: '(' num_expr ',' num_expr ',' num_expr')' ;
mdim_4: '(' num_expr ',' num_expr ',' num_expr ',' num_expr')' ;
```

Blocchi, blocchi condizionali e cicli

Blocchi di codice

Un elemento essenziale per il linguaggio è la rappresentazione di blocchi di codice. Un blocco è costituito da un insieme di linee descrittive racchiuse dalla coppia di parentesi graffa aperta ('{') e chiusa ('}'). In corrispondenza con l'apertura di un blocco di codice, nell'interprete viene creato un nuovo contesto per la definizione di variabili che termina la propria esistenza nel punto in cui il blocco viene chiuso.

```
body_block //un blocco di codice racchiuso tra parentesi graffe
    : '{'
        (body_statement)*
      '}'
    ;

body_statement //le istruzioni che possono essere inserite in un
    blocco di codice
    : fun_call_noret | func_return | sub_def | for_loop |
        conditional_block | object | var_statement ';'
    ;
```

Blocchi condizionali

I blocchi condizionali permettono di subordinare l'esecuzione del codice di un blocco al verificarsi di una condizione stabilita attraverso una espressione booleana. Opzionalmente è possibile specificare un secondo blocco di codice da eseguire nel caso in cui la condizione non sia verificata.

```
conditional_block
    : 'if''(' bool_expr  ')'
        body_block
       ( 'else'
        body_block
       )?
    ;
```

Cicli

Tramite l'uso dei cicli è possibile eseguire blocchi di codice per un numero prefissato di ripetizioni. La definizione di un ciclo *for* prevede di indicare una variabile di controllo seguita da una lista di valori numerici. La variabile di controllo assumerà, uno per ogni passo di ripetizione del blocco, tutti gli elementi presenti nella lista.

```
for_loop
```

```
    : 'for' identifier 'in' numlist
        body_block
    ;
```

Definizione e chiamata di funzioni

La possibilità di definire funzioni parametrizzate da poter richiamare all'interno del proprio codice permette la realizzazione di librerie di codice di utilità generale oltre ad incentivare la scrittura di codice riutilizzabile.

Definizione di funzioni

La definizione di una funzione è composta come prima cosa dalla specifica del tipo di valore restituito (o void nel caso di procedura senza valore di ritorno), seguita dalla parola chiave *function*, dall'identificatore corrispondente al nome della funzione e dalla lista dei parametri di ingresso della funzione racchiusa da una coppia di parentesi tonde aperta e chiusa. La specifica di ogni parametro è costituita dal tipo seguito dall'identificatore con cui la variabile sarà accessibile all'interno del corpo della funzione. Di seguito deve essere specificato il blocco di codice eseguito dalla funzione, al cui interno può essere utilizzata l'istruzione *return* per uscire dalla funzione e specificarne se previsto il valore di ritorno.

```
sub_def //definizione di una funzione
  : 'function' return_type identifier '(' param_def_list ')'
      body_block
    ;

return_type //tipo di valore restituito dalla funzione
  : 'void' | 'string' | 'num' | 'boolean' ;

param_def_list //definizione della lista dei parametri accettati dalla
     funzione
    : (param_def (',' param_def)* )?
    ;

param_def //definizione di un parametro accettato dalla funzione
    : 'num' identifier
    | 'string' identifier
    | 'boolean' identifier
    ;

func_return //esce dalla funzione restituendo il valore specificato
```

```
    : 'return' ( ';' //void
     | num_expr';'
     | str_expr';'
     | bool_expr';'
    )
    ;
```

Chiamata di funzioni

La chiamata di una funzione precedentemente definita è composta dall'istruzione *call* seguita dall'identificatore della funzione da eseguire e dalla lista dei valori dei parametri previsti. Se la funzione prevede la restituzione di un valore, la chiamata può essere inserita come elemento di una espressione.

```
fun_call // chiamata a funzione
    : 'call' identifier '(' param_list ')'
    ;

param_list //lista dei parametri passati alla funzione
    : (param (',' param )* )?
    ;

param
    : num_expr
    | str_expr
    ;
```

4.3.2 Ambiente di simulazione

La descrizione dell'ambiente in cui viene eseguita la simulazione prevede l'uso dell'istruzione *environment* (o abbreviato *env*) seguita dal tipo di ambiente descritto (*wallrect* o *room*) seguito da un blocco per la specifica di tutti i parametri previsti per il tipo di ambiente rappresentato.

```
env_desc //descrizione dell'abiente di simulazione
    : ('environment'|'env') env_type
    ;

env_type_desc //tipi di ambiente supportati
    :
      ( wallrect
```

```
      | room )
    ;

wallrect // recinto rettangolare 'outdoor'
    :
      ('wallrect') wallrect_block
    ;

room  // stanza a base rettangolare
    :
      ('room') wallrect_block
    ;

wallrect_block
    : (';' | '{'(env_param)*'}')
    ;

env_param
    : (width | lenght | height | skymat | gndmat | wallmat | ceilmat |
         erp | cfm | gravity)';'
    ;

width // larghezza
    : 'width' num_expr
    ;

lenght // lunghezza
    : 'lenght' num_expr
    ;

height // altezza
    : 'height' num_expr
    ;

skymat // materiale (texture) utilizzato per la rappresentazione del
    cielo
    : 'sky' str_expr
    ;

gndmat // materiale per il pavimento
    : 'ground' str_expr
    ;
```

```
wallmat // materiale per i muri
    : 'walls' str_expr
    ;

ceilmat // materiale per il soffitto
    : 'ceiling' str_expr
    ;

erp // error reduction parameter globale
    : 'erp' num_expr
    ;

cfm // constraint force mixing globale
    : 'cfm' num_expr
    ;

gravity // forza di gravità
    : 'gravity' num_expr
    ;
```

4.3.3 Oggetti e gruppi di oggetti

Elementi primitivi

Un elemento primitivo viene descritto specificando l'identificatore corrispondente al nome dell'oggetto, seguito da un blocco di descrizione dell'oggetto. Se l'oggetto da inserire è di tipo statico (non si può spostare) è necessario anteporre l'istruzione *static* al suo identificatore.

```
element
    : ('static')? identifier element_block
    ;

element_block //blocco di descrizione dell'oggetto
    : '{'
        (object_statement)*
      '}'
    ;

object_statement
    : (position | quaternion | rotation_axis | material | customgeom |
         shapedef | var_statement | density | body_attached_sensor)';'
```

```
    ;

shapedef //specifica della forma e dimensioni dell'oggetto
    : 'shape' shape
    ;

customgeom // permette di specificare per l'oggetto simulato una forma
     diversa da quella rappresentata visivamente (utile nel caso di
    mesh3d troppo pesanti)
    : 'customgeom' shape
    ;

shape //i tipi primitivi supportati
    : (sphere|box|cylinder|capsule|mesh)
    ;

sphere //parametri per la sfera: raggio
    : 'sphere' mdim_1
    ;

box //parametri per il parallelepipedo: width, height,length
    : 'box' mdim_3
    ;

cylinder //parametri per il cilindro: radius, length
    : 'cylinder' mdim_2
    ;

capsule //parametri per la capsula: radius, length
    : 'capsule' mdim_2
    ;

mesh //parametri mesh definita dall'itente: nomefile, fattore di scala
    : 'mesh''('str_expr ',' num_expr ')'
    ;

position //posizione dell'oggetto (x,y,z)
    : ('position' | 'pos') mdim_3
    ;

quaternion //quaternione di rotazione dell'oggetto
    :
      ('quaternion' | 'quat') mdim_4
```

```
    ;

rotation_axis //rotazione dell'oggetto di ang radianti lungo l'asse:
    parametri ax,ay,az,angle
    : ('rotation_axis' | 'rot_axis') mdim_4
    ;

density //densità dell'oggetto
    : 'density' num_expr
    ;

material //materiale (texture e relative proprietà) applicate all'
    oggetto
    : 'material' str_expr
    ;
```

Contenitori di oggetti

La rappresentazione di gruppi logici di oggetti primitivi è consentita dall'oggetto contenitore virtuale. Un contenitore viene descritto inserendo la parola chiave *container* seguita dall'identificatore che ne rappresenta il nome e il blocco di codice dal quale vengono generati tutti gli elementi del gruppo.

```
container
    : 'container' identifier body_block
    ;
```

Oggetti composti

Un oggetto composto costituisce l'unione fisica e inseparabile di più oggetti elementari che nella simulazione verranno considerati come un blocco unico. La descrizione di un elemento composto inizia con l'utilizzo della parola chiave *composed* seguita dall'identificatore che ne rappresenta il nome e il blocco di codice dal quale vengono prodotti gli elementi che compongono l'oggetto. Il caso di oggetto composto non mobile viene rappresentato anteponendo la parola *static* alla descrizione.

```
composed //oggetto composto
    : ('static')? 'composed' identifier
       composed_element_block
    ;
```

```
composed_element_block //istruzioni del blocco di codice di un
    elemento composto possono essere istruzioni per la descruzione dei
    parametri dell'oggetto di base o codice per la generazione degli
    altri oggetti.
    : '{'
        (object_statement | body_statement)*
      '}'
    ;
```

4.3.4 Giunti tra oggetti

La rappresentazione dei vincoli fisici tra gli oggetti della simulazione avviene tramite l'uso dei giunti. Per descrivere un giunto tra due oggetti è necessario specificare il tipo di giunto (i tipi supportati sono *JointBall*, *JointHinge*, *JointSlider*, *JointUniversal*, *JointHinge2*, *JointFixed*, *JointPR*, *JointPU*, *JointPiston*) seguito dalla coppia di identificatori degli oggetti da collegare racchiusa tra parentesi tonde e da un blocco per la descrizione dei parametri del giunto.

```
joint //specifica di un giunto tra due oggetti
    : joint_type '(' identifier ',' identifier ')'
        joint_block
    ;

joint_type //tipi di giunto supportati
    : 'JointBall' | 'JointHinge' | 'JointSlider' | 'JointUniversal' |
        'JointHinge2'| 'JointFixed' | 'JointPR' | 'JointPU' | '
        JointPiston'
    ;

joint_block //blocco di descrizione dei parametri del giunto
    : ';' | '{'(joint_param)* '}'
    ;

joint_param //i parametri disponibili per i giunti (è possibile
    inserire anche attuatori che agiscano sul giunto)
    : (axis | axis2 | anchor | joint_erp | joint_cfm | actuator)';'
    ;

axis //impostazione dell'asse del giunto
    : 'axis' n=mdim_3
    ;
```

```
axis2 //impostazione del secondo asse del giunto (se previsto)
    : 'axis2' n=mdim_3
    ;

anchor //impostazione del punto di ancoraggio del giunto
    : 'anchor' n=mdim_3
    ;

joint_erp //error reduction parameter specifico per il giunto
    : 'erp' e=num_expr
    ;

joint_cfm //constraint force mixing specifico per il giunto
    : 'cfm' e=num_expr
    ;
```

4.3.5 Descrizione dei robot

Robot

Un elemento chiave per il simulatore è la rappresentazione dei robot, dei sensori e degli attuatori che li compongono. La descrizione di un robot è costituita da due parti. La prima consiste nella specifica dei programmi che agiscono sul robot mentre la seconda è la descrizione degli elementi che ne costituiscono il corpo.

Per descrivere un robot è necessario specificare la parola chiave *robot* seguita dall'identificatore con cui verrà chiamato l'oggetto creato e da un blocco (come sempre racchiuso tra parentesi graffe) contenente nell'ordine indicazioni sul programma da utilizzare per il robot e la descrizione del corpo. La specifica del programma del robot avviene utilizzando la parola chiave *code* seguita dal nome del file progetto contenente tutte le informazioni necessarie alla compilazione dei sorgenti. La descrizione del corpo avviene inserendo parola chiave *body* seguita da un blocco di codice di descrizione che produrrà gli elementi (oggetti primitivi e composti, giunti, sensori, attuatori) che lo compongono.

```
robot
    : 'robot' identifier '{'
          robot_code
          robot_body
      '}'
```

```
    ;

robot_code // indicazione del file progetto per il programma del robot
    : 'code' ( '{' stringval '}' | stringval ';'  )
    ;

robot_body //specifica degli oggetti che compongono il corpo del robot
    : 'body' '{'
        (body_statement)*
      '}'
    ;
```

Sensori

La descrizione di un sensore può essere inserita solamente all'interno di un blocco di descrizione del corpo di un robot. I vari tipi di sensore supportati possono essere suddivisi in due categorie principali, ovvero quelli che devono essere collegati ad un qualche elemento del corpo del robot e quelli indipendenti. I sensori indipendenti possono essere specificati in qualsiasi punto del blocco di descrizione del corpo, mentre quelli che devono essere collegati vanno inseriti all'interno del blocco di descrizione dell'oggetto al quale fanno riferimento. La descrizione di un sensore avviene specificando la parola chiave corrispondente al tipo di sensore, seguita dal nome con il quale verrà identificato e opzionalmente un blocco per la descrizione dei parametri del sensore.

```
body_attached_sensor //specificato all'interno di un oggetto
    : body_attached_sensor_type identifier (sensor_block)?
    ;

standalone_sensor //sensore indipendente da altri elementi
    : standalone_sensor_type identifier (sensor_block)?
    ;

body_attached_sensor_type //sensore collegato ad un oggetto fisico
    : 'SimpleGPS' | 'Compass' | 'LaserDistance' |'SimpleAccelerometer'
    ;

standalone_sensor_type
    : 'Keyboard'
    ;

sensor_block //corpo di descrizione dei parametri di un sensore
```

```
    : '{'
        (sensor_statement)*
      '}'
    ;
```

Attuatori

Gli attuatori installati nei robot permettono di agire sui giunti che ne collegano gli elementi. La descrizione di un attuatore deve quindi essere inserita all'interno del blocco di descrizione delle proprietà del giunto al quale fanno riferimento. Il formato per la descrizione di un attuatore è analogo a quello usato per rappresentare i sensori, composto dal tipo di giunto seguito dall'identificatore e opzionalmente il blocco di descrizione delle proprietà.

```
actuator //specificato all'interno di un giunto
    :  actuator_type identifier (actuator_block)?
    ;

actuator_type //tipi di attuatore supportati
    : 'SimpleMotor' | 'SimpleServo'
    ;

actuator_block //corpo di descrizione dei parametri di un attuatore
    : '{'
        (actuator_statement)*
      '}'
    ;
```

Capitolo 5

Esempi di simulazione

In questo capitolo vengono presentati degli esempi di codice scritto in linguaggio descrittivo per dimostrare le possibilità offerte dal simulatore e soprattutto come semplice riferimento per la descrizione di scenari utilizzando il linguaggio descritto nel precedentemente capitolo.

5.1 Frammenti di codice d'esempio

In questa prima parte vengono presentati piccoli frammenti di codice descrittivo per fornire esempi sull'utilizzo delle componenti di base del linguaggio, a partire dalla definizione di variabili fino alla creazione di funzioni per la generazione di elementi complessi.

5.1.1 Elementi di base

Il primo semplice esempio fornisce una dimostrazione degli elementi di base del linguaggio, dalla definizione di diversi tipi di variabile, a operazioni tra numeri e tra stringhe, all'utilizzo di costanti. Nel codice viene sono inoltre presenti esempi di ciclo ed un blocco condizionale if-else. Nell'esempio vengono calcolati in due modi differenti i numeri dispari da uno fino al numero assegnato alla variabile *a*. Nel primo caso viene utilizzato un ciclo for che parte da 1 e arriva fino alla variabile *a* compresa incrementando ad ogni passo di due il valore della variabile di controllo. Nel secondo caso il ciclo incrementa la variabile di uno e viene utilizzato un blocco condizionale per sommare solo i numeri che non risultano divisibili per due. Nella parte finale vengono mostrate alcune operazioni tra stringhe e viene utilizzato un blocco if-else per controllare che i risultati ottenuti nei

due modi differenti coincidano. Nel codice viene inoltre mostrato l'uso delle due diverse modalità previste per l'inserimento di commenti.

```
/*-----------------------------------
  COMMENTO MULTILINEA
-----------------------------------*/

num a=10;
num ris_1=0;

//metodo 1
for i in {1>=a|2} {
  ris_1=ris_1+i;
}

//metodo 2
num ris_2=0;
for j in {1>=a} {
  if(mod(j,2)!=0) {
    print_num j;
    ris_2=ris_2 + j;
  }
}

string str1="[".string(ris_1)."]";
string str2="[".string(ris_2)."]";

if(ris_1!=ris_2) {
  print_str "Errore: i due risultati sono differenti: ris1=".str1." ris2= ".
      str2;
} else {
  print_str "Il risultato e' ".str1;
}
```

5.1.2 Definizione e chiamata a funzioni

Di seguito un semplice esempio di definizione di funzione e successiva chiamata. Nel codice viene realizzata una funzione ricorsiva per il calcolo del fattoriale di un numero. La funzione viene successivamente chiamata ed il risultato assegnato ad una variabile numerica viene poi stampato in un messaggio di log.

```
function num fattoriale(num N) {
  if(N) {
     return N * (call fattoriale(N-1));
  } else {
     return 1;
  }
```

```
}

num i=call fattoriale(4);

print_str "RISULTATO: ".string(i);
```

5.1.3 Generazione di oggetti

L'utilizzo delle funzioni consente la creazione di codice generativo parametrizzato. A titolo di esempio viene presentato il codice per la generazione di una pila di oggetti secondo uno schema "a piramide" partendo da una base con un numero di elementi parametrizzato e ridotto ad ogni livello di un elemento. La funzione viene chiamata tre volte per generare pile a piramide con di tre dimensioni diverse.

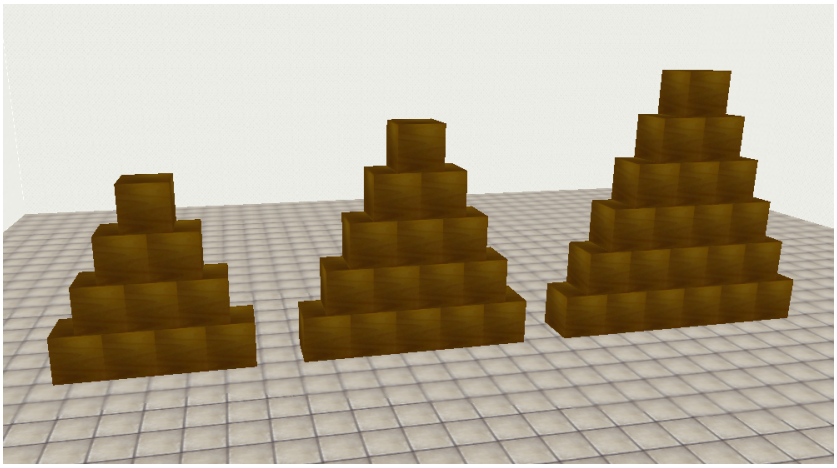

Figura 5.1: L'esempio richiama la funzione per la generazione del muro a piramide con tre parametri diversi.

```
function void piramide(num lato, num n_base,string mat,num start_x,num
    start_z) {
  num x=start_x;
  num y=lato/2;
  num z=start_z;
  string pirname="piramide_".string(start_x).string(start_z);
  for liv in {0>n_base} {
   for i in {0>n_base-liv} {
     id:[pirname.string(i)."_".string(liv)] {
        shape box(lato,lato,lato);
        pos(x+lato*i,y,z);
```

```
          material mat;
        }
      }
      x=x+lato/2;
      y=y+lato;
    }
  }

  call piramide(2, 4, "Legno" ,10,30);
  call piramide(2, 5, "Legno" ,20,30);
  call piramide(2, 6, "Legno" ,32,30);
```

5.1.4 Oggetti e giunti

La rappresentazione di vincoli fisici tra oggetti elementari avviene tramite l'utilizzo di giunti. Per descrivere un giunto è necessario specificare il tipo, seguito dalla coppia di elementi ai quali il vincolo fa riferimento. I parametri supportati dal tipo di giunto rappresentato possono essere inseriti all'interno del blocco di codice che segue la definizione del giunto. Nell'esempio seguente viene rappresentato un semplice carro a quattro ruote, composto da una base rettangolare alla quale sono collegate tramite giunti a cerniera quattro ruote (ottenute come forma complessa caricata da file mesh).

```
function void crea_carretto(num x, num y, num z) {
   num lung=3;
   num larg=8;
   num alt=6;
   num rot=pi/2;

   base {
    shape box(lung,0.5,larg);
    pos(x,y,z);
    rotation_axis(1,0,0,rot);
    material "Legno";
   }

   num spess=0.7;
   num r=0.85;
   num K1=x-(lung/2-r);
   num K2=x+(lung/2+r);

   ruota1 { shape mesh("wheel.mesh",0.01); customgeom cylinder(r,spess);
     rotation_axis(0,1,0,-pi/2); pos(x-(lung/2+spess),y,z-(larg/2-spess));}
   ruota2 { shape mesh("wheel.mesh",0.01); customgeom cylinder(r,spess);
     rotation_axis(0,1,0,pi/2); pos(x+(lung/2+spess),y,z-(larg/2-spess));}
```

```
    ruota3 { shape mesh("wheel.mesh",0.01); customgeom cylinder(r,spess);
      rotation_axis(0,1,0,-pi/2); pos(x-(lung/2+spess),y,z+(larg/2-spess));}
    ruota4 { shape mesh("wheel.mesh",0.01); customgeom cylinder(r,spess);
      rotation_axis(0,1,0,pi/2); pos(x+(lung/2+spess),y,z+(larg/2-spess));}

    JointHinge(base,ruota1) {
       Anchor(x-(lung/2+spess),y,z-larg/2-spess); Axis(1,0,0); }
    JointHinge(base,ruota2) {
       Anchor(x+(lung/2+r),y,z-larg/2-r); Axis(1,0,0); }
    JointHinge(base,ruota3) {
       Anchor(x-(lung/2+r),y,z+(larg/2-r)); Axis(1,0,0); }
    JointHinge(base,ruota4) {
       Anchor(x+(lung/2+r),y,z+(larg/2-r)); Axis(1,0,0); }
  }

call crea_carretto(40,1,20);
```

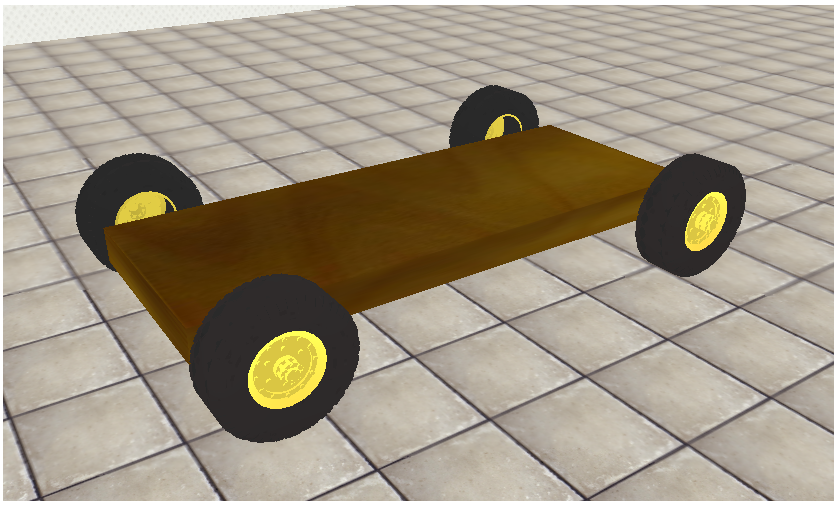

Figura 5.2: Il risultato del file di descrizione del carro a quattro ruote.

5.2 Descrizione dell'ambiente di simulazione

La descrizione dell'ambiente in cui avviene la simulazione è costituita da un preambolo obbligatorio che descrive il tipo di ambiente, specificando le dimensioni e i materiali da utilizzare per la rappresentazione delle singole componenti dell'ambiente scelto. Di seguito è possibile specificare in via opzionale la descrizione degli elementi aggiuntivi che faranno parte dell'ambiente iniziale. Nell'esempio di seguito viene presentata la descrizione di un ambiente interno, una stanza, nella quale sono introdotti

una serie di elementi decorativi. Gli oggetti inseriti sono di tipo statico e dunque non potranno essere spostati durante la simulazione.

```
/*********************************************************
  ESEMPIO FILE DI DESCRIZIONE DELLE SCENE
*********************************************************/

environment room {
  width 60;
  lenght 50;
  height 20;
  sky "Black";
  ground "Piastrelle2";
  walls "Pattern2";
  ceiling "MuroRigheRosso";
  gravity -9.8;
}


static QUADRO {
  shape box(0.1,5.25,7);
  pos(0,12,20);
  material "Test2D";
}

static POSTER {
  shape box(0.1,10.3,7);
  pos(60,9,15);
  material "Schema";
}

static SEDIA {
  shape mesh("sedia.mesh",0.016);
  customgeom box(10,10,10);
  rotation_axis(1,0,0,pi/2);
  density 0.1;
  pos(18,0,12);
}

static SCRIVANIA {
  shape mesh("scrivania.mesh",0.07);
  customgeom box(10,10,10);
  rotation_axis(1,0,0,pi/2);
  density 0.1;
  pos(20,0,1);
}

static IMAC {
  shape mesh("imac.mesh",10);
```

```
  customgeom box(5,5,5);
  rotation_axis(1,0,0,pi/2);
  density 0.1;
  pos(22,7,2.5);
}

static VENTILATORE {
  shape mesh("ventilatore.mesh",0.2);
  customgeom box(10,10,10);
  rotation_axis(1,0,0,0);
  density 0.1;
  pos(20,20,30);
}


static APPENDIABITI {
  shape mesh("appendiabiti.mesh",8);
  customgeom box(2,10,2);
  density 0.1;
  pos(53,0,4);
}


static PORTA {
  shape mesh("door.mesh",7);
  customgeom box(5,10,0.5);
  rotation_axis(1,0,0,0);
  density 0.1;
  pos(40,0,0.1);
}

static FILE_CABINET {
  shape mesh("file_cabinet.mesh",0.25);
  customgeom box(5,2,3);
  rotation_axis(1,0,0,0);
  density 0.1;
  pos(5,0,2);
}

static PIATTAFORMA {
  shape box(12,12,0.2);
  pos(48,0,33);
  material "BasePiattaforma";
}

static CARTELLO_ATTENZIONE {
  shape box(0.1,2,2.18);
  pos(60,7,33);
  material "AttenzioneRobot";
```

```
}
```

Figura 5.3: Il risultato del file di descrizione dell'ambiente.

5.3 Realizzazione di un semplice robot

La rappresentazione di un robot all'interno del simulatore è suddivisa in due parti, una costituita dal file descrittivo per la rappresentazione degli elementi che costituiscono il corpo, l'altra comprendente il file progetto e i file sorgente contenenti il programma da eseguire sul robot.

5.3.1 La descrizione del corpo del robot

La descrizione del corpo del robot avviene come una qualsiasi composizione di oggetti elementari connessi da giunti, sui quali possono essere inseriti sensori ed attuatori. La descrizione del robot dovrà inoltre presentare il nome del file progetto da utilizzare per il programma.

Di seguito viene presentato il codice descrittivo per la realizzazione di un semplice robot cilindrico a due ruote. Il corpo è costituito da una carrozzeria cilindrica, sulla quale sono connesse tramite giunti a cerniera le due ruote e due perni sferici per mantenere l'equilibrio. Nel blocco

centrale sono introdotti sensori per la rilevazione della posizione e dell'accelerazione, mentre sui giunti che vincolano le ruote agiscono due motori.

Nella parte finale viene associato al robot il corrispondente file progetto da utilizzare per la compilazione del programma da eseguire.

```
function void crea_robot_tank(num x, num y, num z) {

 string base_name="base_tankbot_".string(x).string(y).string(z);

  num r_corpo=3.2; //raggio del corpo robot
  num r_ruota=0.85; //raggio della ruota

  num h_corpo=3.0;

  num delta_y_ruota=0.2;
  num y_corpo=y+(h_corpo/2)+r_ruota-delta_y_ruota;

  Keyboard kbd;

  composed id:[base_name]{
   pos(x,y_corpo,z);

   corpo_robot
    {
      shape mesh("robot_hq.mesh",0.30);
      customgeom cylinder(3.0,3.2);
      rotation_axis(1,0,0,-pi/2);
      density 0.05;
      pos(0,0,0);
    }

    cubo_centrale {
       shape box(1,1,1); pos(0,0,0); material "Red50";
       SimpleGPS GPS1; SimpleAccelerometer ACC; }
  }

  num d_ruota=0.1;
  num yr=y+r_ruota;
  num z1=z+(r_corpo+d_ruota);
  num z2=z-(r_corpo+d_ruota);
  ruota_1 { shape mesh("wheel.mesh",0.01); customgeom cylinder(0.85,0.7);
    pos(x,yr,z1); density 0.3; }
  ruota_2 { shape mesh("wheel.mesh",0.01); customgeom cylinder(0.85,0.7);
    pos(x,yr,z2); rotation_axis(0,1,0,pi); density 0.3;  }

  num dim_perno=0.3;//(r_ruota-delta_y_ruota)/2;
  num delta_p=r_corpo*0.8;
  perno_1 { shape sphere(dim_perno);
```

```
    pos(x+delta_p,y+dim_perno,z);  density 0.3; material "Black"; }
  perno_2 { shape sphere(dim_perno);
    pos(x-delta_p,y+dim_perno,z); density 0.3; material "Black"; }

  JointHinge(id:[base_name],ruota_1) {
    Anchor(x,yr,z1); Axis(0,0,1);  SimpleMotor Motor1; }
  JointHinge(id:[base_name],ruota_2) {
    Anchor(x,yr,z2); Axis(0,0,1);  SimpleMotor Motor2; }

  JointBall(id:[base_name],perno_1) { Anchor(x+delta_p,y+dim_perno,z);}
  JointBall(id:[base_name],perno_2) { Anchor(x-delta_p,y+dim_perno,z);}
}

robot tankbot {
  code "/working/tesi/trunk/plugins/example/Cpp/tankbot.rprj";
  body {
    call crea_robot_tank(30,0,20);
  }
}
```

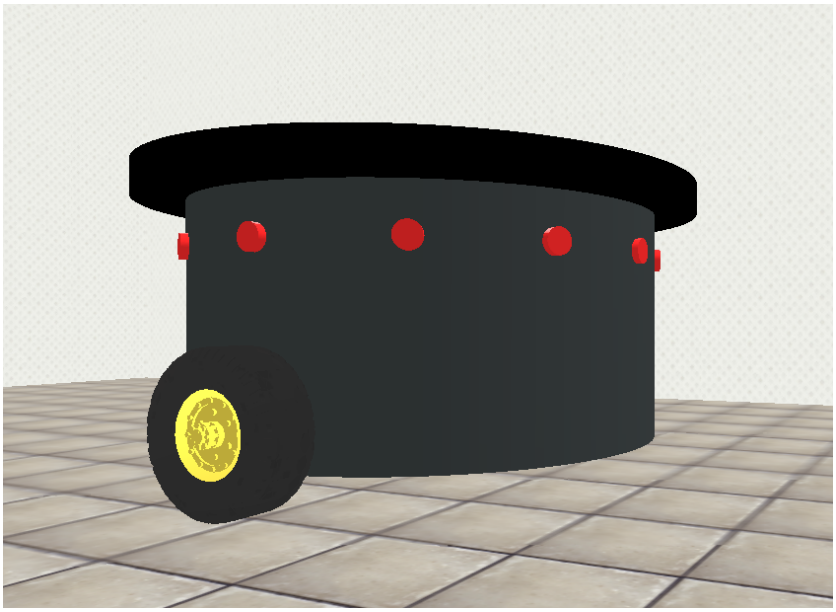

Figura 5.4: Il robot a due ruote visualizzato nel simulatore.

5.3.2 Il programma del robot

I programmi dei robot forniti come file sorgente in fase di inserimento nel simulatore possono accedere all'ambiente virtuale tramite le interfacce per i sensori e gli attuatori previste dalla RobotAPI.

Di seguito viene presentato un semplice programma in C++ per mostrare l'utilizzo delle interfacce previste per i sensori e gli attuatori installati nel robot precedentemente descritto.

```
#include <plugins/robot_software_api.h>


class PLUGIN_EXPORT RobotSoftwareTest1 : public RobotSoftware {
public:

  RobotSoftwareTest1(Robot *r) : RobotSoftware(r)
  { dir=1; }

  virtual void preStep() {

      dReal SPEED=0;

      RemoteControl *kb=_robot->getSensorInstance<RemoteControl>("kbd");
      if(kb) {
        int val=0;
        bool res=kb->getValue(string("I"),val);
        if(res && val!=0) {
           dir=-dir;
        }
      }

      GPSSensor *gps=_robot->getSensorInstance<GPSSensor>("GPS1");
      if(gps) {
         Vector3D c;
         if(gps->getValue(c)) {
           cout << "GPS: "<< gps->getValueString()<<endl;
           if(dir>0) { if(c.x>50) dir=-dir; }
           else { if(c.x<10) dir=-dir; }
           SPEED=dir*10;
         } else { cout << "ERROR READING GPS VAL"<<endl; }
      } else cout << "ERROR GETTING GPS"<<endl;

      Accelerometer *acc=_robot->getSensorInstance<Accelerometer>("ACC");
      if(acc) {
        Vector3D a;
        if(acc->getValue(a)) {
          cout << "ACCELERATION: x="<<a.x<<" y="<<a.y<< " z="<<a.z<<endl;
        }
      } else cout << "Errore nell'accesso all'accelerometro"<<endl;

      Motor*motor1=_robot->getActuatorInstance<Motor>("Motor1");
      Motor*motor2=_robot->getActuatorInstance<Motor>("Motor2");
      if(motor1 && motor2) {
```

```
        motor1->setSpeed(SPEED);
        motor2->setSpeed(SPEED);
      } else cout << "Errore nell'accesso ai motori!"<<endl;
  }

protected:

  int dir;

};

ROBOT_SOFTWARE_PLUGIN(RobotSoftwareTest1,"ProgrammaEsempio")
```

Assieme al file sorgente è necessario specificare un file di progetto, che ne indichi il tipo, i file sorgente da includere nella compilazione ed eventuali parametri dipendenti dal plugin di compilazione utilizzato. Di seguito viene riportato il file di progetto utilizzato per il programma appena descritto.

```
#PROGETTO DI ESEMPIO
TYPE:CPP
SRC:tankbot.cpp
INCLUDE_DIR:/working/tesi/trunk/
```

Capitolo 6

Conclusioni

6.1 Gli obiettivi realizzati

L'obiettivo della presente tesi consiste nella realizzazione di un ambiente di simulazione in grado di consentire la predisposizione di laboratori virtuali condivisibili in rete sui quali effettuare esperimenti nel campo della robotica.

I requisiti iniziali sono stati in parte ricavati dagli obiettivi realizzati dal progetto VLAB, integrati dopo una analisi dello stato dell'arte del campo dei simulatori impiegati nella robotica.

Il risultato del lavoro di progettazione e sviluppo è stata la realizzazione di un ambiente software programmato in C++ dotato delle caratteristiche richieste:

- in grado di rappresentare i robot, i sensori e gli attuatori che li costituiscono;
- basato su un'architettura client-server per consentire l'accesso alla simulazione attraverso la rete;
- in grado di fornire una rappresentazione tridimensionale dei risultati della simulazione;
- dotato di un opportuno linguaggio di descrizione degli elementi appositamente progettato;
- in grado di ricevere dagli utenti i programmi per i robot, compilarli ed eseguirli durante la simulazione;
- con supporto estensibile tramite utilizzo di plugin alla compilazione ed esecuzione di programmi scritti in altri linguaggi di programmazione.

6.2 Sviluppi futuri

Il sistema risultante dal lavoro di progettazione e sviluppo per questa tesi di laurea costituisce un ambiente integrato utilizzabile negli ambiti previsti e con un'architettura pensata anche per favorirne l'estensibilità e lo sviluppo da parte dell'utente di librerie di codice descrittivo riutilizzabili. Pur comprendendo nel risultato finale le numerose funzionalità e componenti previste, è stato necessario limitare l'attenzione su alcuni aspetti che si possono comunque prestare ad ulteriori lavori di sviluppo.

Un primo punto sul quale si potrebbe concentrare l'attenzione è quello di dotare il motore di simulazione di moduli per la simulazione aggiuntivi oltre a quello per la gestione della dinamica dei corpi, per consentire la simulazione di altri campi della fisica, quali ad esempio le trasmissione dei messaggi via radio, diffusione del suono o proprietà termodinamiche del sistema simulato.

Un'ulteriore direzione di sviluppo che potrebbe essere seguita per lavori futuri è l'ampliamento della dotazione e delle caratteristiche supportate dei sensori e degli attuatori da affiancare a quelli di base attualmente previsti.

Infine un aspetto che potrebbe risultare utile studiare più nel dettaglio riguarda la sicurezza del sistema. In primo luogo si potrebbe introdurre un meccanismo per l'autenticazione da parte degli utenti, affiancandolo in seguito ad un sistema che consenta di validare il contenuto dei programmi ricevuti per i robot, permettendo la gestione delle funzionalità ammesse sulla base dell'account dell'utente che invia i file sorgente.

Bibliografia

[1] Andrei Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. C++ in-depth series. Addison-Wesley, 2001. ISBN: 9780201704310.

[2] Stefano Carpin. «Sviluppo di un ambiente condivisibile in rete per robotica simulata». Tesi di laurea. Università di Padova, 1998/1999.

[3] David Goldsman, Richard E. Nance e James R. Wilson. «A brief history of simulation revisited». In: *Winter Simulation Conference*. 2010, pp. 567–574.

[4] B. Gustafsson. *Fundamentals of Scientific Computing*. Texts in Computational Science and Engineering. Springer, 2011. ISBN: 9783642194948.

[5] C.S. Horstmann e G. Cornell. *Core Java: Advanced Features*. Sun Core Series. Prentice Hall, 2008. ISBN: 9780132354790.

[6] Gregory Junker. *Pro Ogre 3d Programming*. The Expert's Voice in Open Source. Apress, 2006. ISBN: 9781590597101.

[7] Jean-Claude Latombe. *Robot Motion Planning:* Kluwer international series in engineering and computer science: Robotics. Kluwer Academic Publishers, 1990. ISBN: 9780792391296.

[8] Roger Mchaney. *Understanding Computer Simulation*. Ventus Publishing ApS, 2009. ISBN: 978-87-7681-505-9.

[9] Robin R. Murphy. *Introduction to Ai Robotics*. Intelligent Robotics And Autonomous Agents. Mit Press, 2000. ISBN: 9780262133838.

[10] James Nutaro. *Building Software for Simulation: Theory and Algorithms, with Applications in C++*. John Wiley & Sons, 2010. ISBN: 9780470414699.

[11] Terence Parr. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. Pragmatic Bookshelf Series. Pragmatic Bookshelf, 2010. ISBN: 9781934356456.

[12] Terence Parr. *The Definitive ANTLR Reference Guide: Building Domain-specific Languages*. Pragmatic Bookshelf Series. Pragmatic Bookshelf, 2007. ISBN: 9780978739256.

[13] M. Reddy. *API Design for C++*. Morgan Kaufmann. Elsevier Science, 2011. ISBN: 9780123850034.

[14] James J. Swain. «Software Survey: Simulation - Back to the future». In: *OR/MS Today* 38.5 (ott. 2011).

Risorse Online

[15] *Java Native Interface 6.0 Specification*. URL: `http://docs.oracle.com/javase/6/docs/technotes/guides/jni/spec/jniTOC.html` (visitato il 14/06/2012).

[16] *The ODE Manual*. URL: `http://ode-wiki.org/wiki` (visitato il 14/06/2012).

[17] *Computer simulation - Wikipedia, The Free Encyclopedia*. URL: `http://en.wikipedia.org/w/index.php?title=Computer_simulation` (visitato il 14/06/2012).

[18] *How to Build a Robot Tutorial - Society of Robots*. URL: `http://www.societyofrobots.com` (visitato il 02/07/2012).

[19] *ANTLR Parser Generator - Home page*. URL: `http://www.antlr.org` (visitato il 14/06/2012).

[20] *Open Dynamics Engine - home page*. URL: `http://www.ode.org` (visitato il 14/06/2012).

[21] *OGRE - Open Source 3D Graphics Engine - Home Page*. URL: `http://www.ogre3d.org` (visitato il 14/06/2012).

[22] *The player project Home page*. URL: `http://playerstage.sourceforge.net/wiki/Main_Page` (visitato il 02/07/2012).

[23] *Gazebo - Home page*. URL: `http://gazebosim.org` (visitato il 02/07/2012).

[24] *USARSim - Home Page*. URL: `http://usarsim.sourceforge.net/` (visitato il 02/07/2012).

[25] *The RoboCup Soccer Simulator*. URL: `http://sserver.sf.net/` (visitato il 02/07/2012).

[26] *Simbad 3D Robot Simulator*. URL: `http://simbad.sourceforge.net` (visitato il 02/07/2012).

[27] *The breve Simulation Environment*. URL: `http://www.spiderland.org/` (visitato il 02/07/2012).

[28] *RP1: The Rossum's Playhouse Mobile-Robot Simulator*. URL: `http://rossum.sourceforge.net/sim.html` (visitato il 02/07/2012).

[29] *Webots robot simulator*. URL: `http://www.cyberbotics.com` (visitato il 02/07/2012).

[30] *anyKode Marilou - Modeling and simulation environment for Robotics*. URL: `http://www.anykode.com` (visitato il 02/07/2012).

[31] *EyeSim - EyeBot Simulator*. URL: `http://robotics.ee.uwa.edu.au/eyebot/doc/sim/sim.html` (visitato il 02/07/2012).

[32] *Microsoft Robotics Developer Studio*. URL: `http://www.microsoft.com/robotics/` (visitato il 02/07/2012).

[33] *Microsoft RDS - Simulation Overview*. URL: `http://msdn.microsoft.com/en-us/library/bb483076.aspx` (visitato il 02/07/2012).

[34] *V-REP Virtual Robot Experimentation Platform*. URL: `http://www.v-rep.eu` (visitato il 02/07/2012).