ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

Corso di Laurea in Ingegneria e Scienze Informatiche

Un sistema di realtà aumentata per la Robotica di sciame

Tesi di laurea in:
Programmazione ad oggetti

Relatore
Prof. Mirko Viroli

Candidato
Marchetti Davide

Correlatore
Dott. Gianluca Aguzzi

IV Sessione di Laurea
Anno Accademico 2024-2025

Sommario

La realtà aumentata rappresenta uno strumento efficace per integrare contenuti digitali all'interno dell'ambiente fisico, consentendo nuove modalità di interazione e visualizzazione dei dati. In questo contesto, la presente tesi descrive lo sviluppo di un'applicazione web di realtà aumentata finalizzata alla sovrapposizione di robot virtuali a robot reali o simulati. L'obiettivo del lavoro è di progettare e implementare un sistema capace di acquisire in tempo reale le posizioni dei robot appartenenti ad uno sciame robotico, gestito dal framework Scala Fields (ScaFi), basato sui principi dell'Aggregate Computing (AC), e renderizzarle all'interno di una scena aumentata accessibile tramite browser e dispositivi dotati di webcam. L'applicazione integra ThreeJS e WebXR per la gestione della scena tridimensionale e delle funzionalità AR. La procedura prevede una fase iniziale di calibrazione dell'arena virtuale mediante il riconoscimento di marker ArUco, inquadrati dalla fotocamera, al fine di allineare correttamente sistema di riferimento reale e digitale. I dati dei robot, trasmessi tramite protocollo MQTT, vengono quindi trasformati dal sistema di coordinate del simulatore a quello dell'arena virtuale, applicando le opportune trasformazioni geometriche per garantire coerenza spaziale. Il contributo principale consiste nella progettazione del modulo di integrazione in realtà aumentata e nella definizione della pipeline di trasformazione e sincronizzazione dei dati. Il sistema è stato validato attraverso test sperimentali, volti a valutare l'accuratezza del posizionamento e ad analizzare le principali fonti di errore, dimostrando la validità e l'efficacia dell'approccio proposto.

Indice

Elenco delle figure

Elenco dei Listati

Capitolo 1

Introduzione

Negli ultimi anni, la Realtà Aumentata (AR) ha acquisito un ruolo sempre più centrale in numerosi ambiti scientifici e tecnologici, grazie all'aumento delle prestazioni dei dispositivi mobili e alla disponibilità di librerie software capaci di gestire contenuti 3D immersivi, nella maggior parte dei *browser* più comuni. La possibilità di sovrapporre elementi virtuali all'ambiente reale ha aperto nuove prospettive sia nel campo della didattica e dell'intrattenimento digitale, che nella robotica e nelle simulazioni industriali, ma anche in campi medici e militari. In particolare, la combinazione tra AR e simulazioni di robot mobili permette di osservare in tempo reale sciami di robot, senza dover ricreare un laboratorio fisico completo, offrendo così un potente strumento di visualizzazione e sperimentazione.

Il presente lavoro di tesi si propone di sviluppare un sistema in grado di inizializzare una scena in realtà aumentata, nella quale vengono mostrati dei robot virtuali, all'interno di un'arena delimitata da *marker ArUco* [RRMSMC18] [GJMSMCMC15] reali, replicando il comportamento simulato da un emulatore, chiamato *robot-emulation* [rob25].
La scena è stata realizzata in *JavaScript*, utilizzando *ThreeJS* [Cab26] per la gestione della grafica 3D e *WebXR* [Wor19] per integrare le funzionalità di realtà aumentata direttamente nel browser. La logica dello sciame, invece, è gestita da un sistema sviluppato in Scala, con l'ausilio del *framework* Scala Fields (ScaFi) [CVAP22], che implementa i principi dell'Aggregate Computing (AC) [BPV15]

per simulare comportamenti emergenti in modo scalabile e modulare. Le posizioni dei robot vengono trasmesse in tempo reale alla scena virtuale tramite Message Queuing Telemetry Transport (MQTT), permettendo all'arena virtuale di rimanere sincronizzata con quella simulata.

Il principale obiettivo del progetto consiste nel creare un ponte tra simulazione digitale ed una scena immersiva, consentendo agli utenti di osservare in tempo reale il comportamento collettivo dei robot direttamente nello spazio fisico circostante. In particolare, il lavoro si pone tre obiettivi principali:

1. **Realizzare un sistema AR stabile e interattivo**, capace di rilevare marker fisici e definire dinamicamente un'arena virtuale.
2. **Integrare la simulazione dello sciame**, aggiornando continuamente le posizioni dei robot all'interno della scena.
3. **Riprodurre fedelmente la scena dell'emulatore**, fornendo un riferimento visivo immediato e comparabile con la simulazione originale.

L'approccio seguito per lo sviluppo del progetto è modulare: la scena 3D si occupa esclusivamente della visualizzazione e dell'interazione, mentre la logica dello sciame è gestita da un sistema esterno. Questo consente di ottenere una maggiore scalabilità e riusabilità del codice, oltre a semplificare la manutenzione futura. L'uso di protocolli standard, come MQTT, garantisce l'affidabilità nella trasmissione dei dati e la compatibilità tra i diversi moduli, ma soprattutto può essere integrato in qualsiasi tipo di sistema compatibile, senza influenzare in profondità l'architettura.

Struttura della tesi Il Capitolo 2 fornisce un'introduzione al contesto del progetto, come Project Emerge e la realtà aumentata; in aggiunta vengono descritte le principali tecnologie utilizzate come ThreeJS, WebXR e MQTT. Segue il Capitolo 3 dove vengono fissati gli obiettivi ed i requisiti da soddisfare durante la realizzazione del progetto. Inoltre viene analizzata l'architettura del sistema sviluppato. All'interno del Capitolo 4 viene descritta in dettaglio l'architettura progettata, giustificando tutte le scelte di sviluppo argomentate nel Capitolo 5. Più

in dettaglio viene mostrata l'implementazione dell'architettura realizzata, con le relative librerie utilizzate. Successivamente nel Capitolo 6 vengono effettuati degli studi sulla stabilità e prestazioni del sistema, mostrando grafici e immagini relative ad ogni analisi svolta. Infine nel Capitolo 7 vengono riassunte le funzionalità del sistema assieme alle rispettive limitazioni, arricchendo il tutto con possibili integrazioni future.

Capitolo 2

Background

2.1 Project Emerge

Project Emerge [pro25] nasce con l'obiettivo di esplorare il coordinamento di uno sciame di droni riproducendo le dinamiche tipiche di un sistema distribuito reale. L'idea alla base del progetto è quella di osservare come un insieme di agenti autonomi, dotati di capacità computazionali e comunicative limitate, possa raggiungere comportamenti collettivi coerenti senza fare affidamento su un controllo centralizzato.
Il progetto propone un'implementazione del nucleo logico dell'AC, adottando Scala come linguaggio di programmazione attraverso l'utilizzo del framework ScaFi. ScaFi è una libreria, scritta in Scala, e framework per per la programmazione aggregata, che implementa una variante della semantica operazionale dell'Higher-Order Field Calculus (HOFC), resa disponibile come Domain Specific Language (DSL), fornendo una piattaforma e un insieme di Application Programming Interface (API) per la simulazione e l'esecuzione di sistemi e applicazioni aggregate. Grazie all'impiego di sistemi decentralizzati, ogni elemento elabora localmente le informazioni e le condivide con i nodi vicini, superando i limiti di un approccio centralizzato, spesso non ideale in contesti dinamici. Applicato ad uno sciame di droni, è possibile definire obiettivi collettivi, come la copertura spaziale, formazione o esplorazione cooperativa, in modo astratto, garantendo l'adattamento automatico alle variazioni topologiche dell'ambiente circostante.

In seguito le simulazioni si sono trasformate in esperienze reali: sono stati realizzati dei droni fisici in grado di coordinarsi all'interno di uno sciame. Ognuno composto da microcontrollori in grado di gestire comunicazioni *wireless*, monitoraggio e gestione energetica in tempo reale ed il movimento del robot, che dipende dalla velocità di entrambe le coppie di ruote motrici. Tramite l'utilizzo di un'interfaccia *web* è possibile visualizzare lo stato di tutti i robot, controllarli individualmente ed eventualmente modificare la formazione corrente dell'interno sciame.
Infine viene proposto un simulatore in *Python*, fondamentale per lo sviluppo di questa tesi, dove in mancanza robot fisici è possibile simularne l'esistenza. Permette di gestire numerosi robot virtuali e modificarne lo stato, utilizzando appositi comandi, inviando costantemente le opportune informazioni al *broker* MQTT. Segue in Figura 2.1 la struttura generale del progetto.

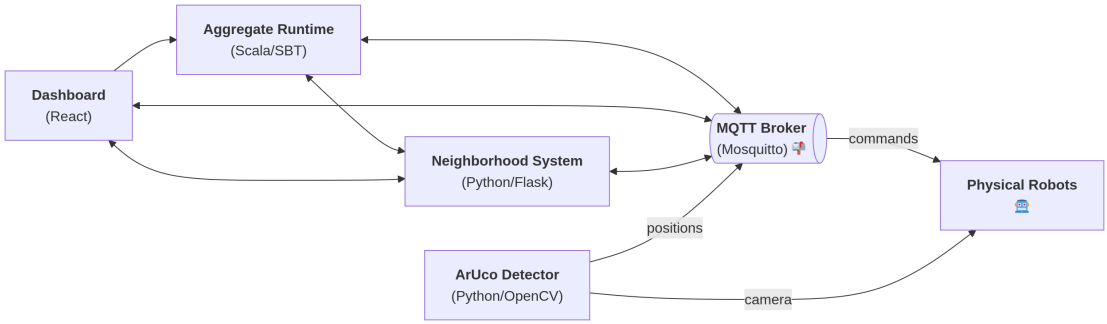


Figura 2.1: Struttura di Project Emerge

2.2 Realtà aumentata

La Realtà Aumentata (AR) è una tecnologia che permette di arricchire il mondo fisico con contenuti digitali, come immagini, informazioni testuali, modelli tridimensionali, integrati in tempo reale. A differenza della Realtà Virtuale (VR), che immerge l'utente in uno spazio interamente simulato sostituendo la percezione dell'ambiente reale, l'AR si caratterizza per la coesistenza tra mondo fisico e componenti digitali, preservando la continuità con lo spazio reale. Tale integrazione arricchisce l'esperienza percettiva senza interrompere l'interazione con l'ambiente circostante, permettendo di realizzare forme di interazione più naturali e conte-

stualizzate.
Dal punto di vista tecnico, un sistema di realtà aumentata si basa su tre componenti fondamentali:

- acquisizione dell'ambiente reale tramite sensori, come fotocamere o sensori di profondità;
- elaborazione e tracciamento spaziale per stimare la posizione e l'orientamento di oggetti reali (*tracking* e *pose estimation*);
- *rendering* in tempo reale degli oggetti virtuali;

Tecniche come il *marker-based* tracking, il *markerless* tracking e gli approcci basati su Simultaneous Localization and Mapping (SLAM), consentono di garantire coerenza geometrica e stabilità visiva dei contenuti aumentati. Dal punto di vista concettuale, un sistema può essere definito di realtà aumentata se soddisfa tre requisiti fondamentali: combinazione di elementi reali e virtuali, interattività in tempo reale e corretta registrazione spaziale tra oggetti digitali e ambiente fisico. Tali caratteristiche rendono l'AR una tecnologia abilitante nell'ambito dell'interazione uomo-macchina, contribuendo allo sviluppo di interfacce sempre più immersive e contestualizzate.

2.3 ThreeJS

ThreeJS è una libreria in JavaScript progettata per semplificare lo sviluppo di applicazioni di grafica tridimensionale in ambienti web. Si basa su WebGL, un'API a basso livello che consente l'accesso diretto alle funzionalità della *GPU* attraverso il browser. L'utilizzo diretto di WebGL richiede una conoscenza approfondita della pipeline grafica e della programmazione tramite *shader*, per questo motivo ThreeJS introduce un livello di astrazione che consente di sviluppare scene tridimensionali complesse riducendo la complessità implementativa, pur mantenendo un elevato grado di flessibilità.
L'architettura di ThreeJS riflette i principali concetti della grafica computazionale tridimensionale e si basa su tre componenti fondamentali: scena, telecamera e *renderer*.

Una scena contiene tutti gli oggetti tridimensionali, le sorgenti luminose e gli elementi di supporto. La telecamera definisce il punto di vista dell'osservatore e può essere di tipo prospettico o ortografico, influenzando il modello di proiezione della scena sul piano di visualizzazione.
Il renderer, generalmente `WebGLRenderer`, si occupa di eseguire la pipeline di rendering sfruttando le funzionalità offerte da WebGL. Gli oggetti rappresentati in una scena, chiamati *mesh*, sono costituiti da due componenti principali:

- **Geometria**: descrive la struttura dei vertici, delle normali e delle coordinate *texture*.
- **Materiale**: definisce le proprietà visive dell'oggetto, texture e il suo comportamento rispetto alle sorgenti luminose.

Nel listato 2.1 viene riportata l'implementazione di una semplice scena 3D con all'interno una telecamera prospettica che inquadra un cubo che ruota nel tempo su due assi diversi, mostrata nella Figura 2.2.

Listato 2.1: Esempio di una scena 3D con ThreeJS

```
import * as THREE from 'three';

const width = window.innerWidth;
const height = window.innerHeight;
// inizializza il renderer con eventuali parametri aggiuntivi
const renderer = new THREE.WebGLRenderer({ antialias: true });
// imposta la grandezza in pixel dell'elemento dove renderizzare la scena
renderer.setSize(width, height);

// parametri per una telecamera con proiezione prospettica
const FOV = 45;
const NEAR = 0.1;
const FAR = 100;
// inizializza la telecamera prospettica
const camera = new THREE.PerspectiveCamera(FOV, width / height, NEAR, FAR);
camera.position.z = 5;

// creazione di una scena
const scene = new THREE.Scene();

// creazione di una mesh con la geometria di un cubo
const geometry = new THREE.BoxGeometry(1, 1, 1);
// creazione di un material di colore rosso
const material = new THREE.MeshBasicMaterial({ color: 0xff0000 });
```

```
25 const mesh = new THREE.Mesh(geometry, material);
26 // aggiunta della mesh alla scena
27 scene.add(mesh);
28 // aggiungo l'elemento creato dal renderer, se non specificato, alla pagina HTML
29 document.body.appendChild(renderer.domElement);
30
31 // definisce quale funzione eseguire ad ogni `requestAnimationFrame()`
32 renderer.setAnimationLoop(animate);
33 function animate(time) {
34     // operazioni sulle mesh ad ogni frame
35     mesh.rotation.x = time / 1000;
36     mesh.rotation.y = time / 1000;
37     // render della scena nel canvas
38     renderer.render(scene, camera);
39 }
```

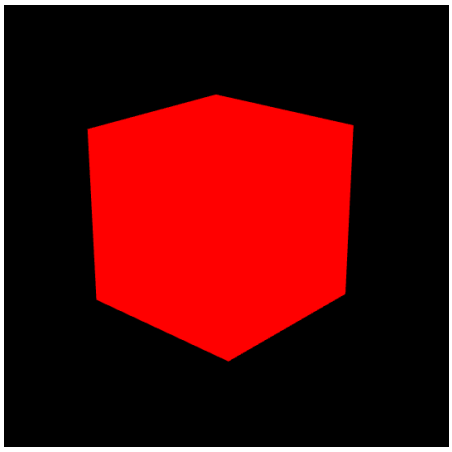


Figura 2.2: Scena 3D di esempio in ThreeJS

Un aspetto di particolare interesse di ThreeJS è il supporto nativo allo standard WebXR, un'API che consente la realizzazione di applicazioni di VR e AR direttamente all'interno del browser. ThreeJS fornisce un livello di integrazione che semplifica la gestione dei dispositivi XR, come visori VR e dispositivi mobili compatibili con AR, astrando la complessità della comunicazione diretta con l'API WebXR. Attraverso l'utilizzo di componenti dedicati, è possibile creare ambienti immersivi tridimensionali in cui la scena viene renderizzata in modo stereoscopico e aggiornata in funzione dei movimenti dell'utente. L'integrazione con WebXR permette inoltre di gestire input avanzati, come *controller* e sistemi di tracciamento.

2.3.1 WebXR

WebXR rappresenta un'evoluzione significativa nel web, consentendo l'accesso diretto ad esperienze di realtà virtuale e aumentata attraverso il browser. Tuttavia, questa tecnologia comporta dei rischi elevati se la sicurezza dell'utente non viene gestita correttamente.
La *WebXR Device API* permette di interagire con numerosi componenti e sensori avanzati come accelerometri, giroscopi, fotocamere e microfoni, e persino dispositivi di input specializzati. Alcune funzionalità non possono essere utilizzate se la connessione non è sicura o se la pagina non è in un contesto *trusted*, proprio per evitare che contenuti malevoli possano compromettere la privacy o la sicurezza del dispositivo.
Per prevenire l'intercettazione o la manipolazione delle informazioni sensibili, tutte le sessioni WebXR devono essere servite tramite connessioni sicure *HTTPS*. Si basa su Transport Layer Security (TLS), garantendo la codifica *end-to-end* dei dati tra client e server, l'autenticità del sito web e l'integrità dei contenuti, impedendo attacchi di tipo *man-in-the-middle* o *spoofing*.
Il browser non permette l'inizializzazione di sessioni immersive su pagine non sicure e interrompe l'esecuzione di operazioni non approvate dall'utente. Le quali, riprendono la normale esecuzione nel momento in cui l'utente acconsente all'accesso agli eventuali sensori e dati sensibili condivisi.

2.4 MQTT

MQTT è un protocollo di comunicazione, divenuto standard OASIS [BBBG19], basato sul paradigma *publish-subscribe* con l'ausilio di un protocollo di trasporto bidirezionale e affidabile. Progettato per facilitare la comunicazione all'interno di reti instabili o tra dispositivi che utilizzano un numero limitato di risorse. In questo modello sono necessari tre componenti principali, come mostrato in Figura 2.3:

- **broker**: nodo centrale con il compito di distribuire i messaggi;
- **publisher**: dispositivo o applicazione che pubblica messaggi;
- **subscriber**: dispositivo o applicazione che riceve messaggi;

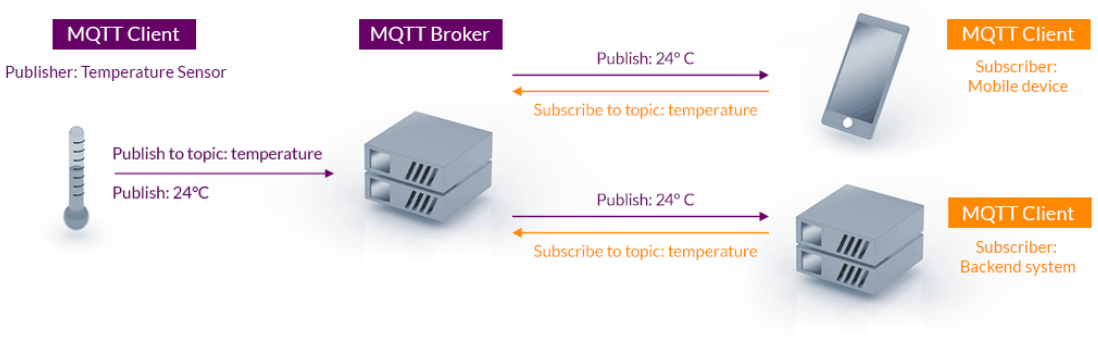


Figura 2.3: Esempio di comunicazione utilizzando MQTT

Più nello specifico, ogni messaggio inviato e ricevuto viene organizzato in stringhe gerarchiche, chiamate *topic*. Ogni publisher pubblica un messaggio in un topic e un subscriber riceve i messaggi dai topic a cui viene sottoscritto. L'operazione di filtraggio dei messaggi nei relativi topic viene svolta dal broker.
L'architettura di MQTT introduce diversi disaccoppiamenti tra mittenti e destinatari:

- **Disaccoppiamento spaziale**: ogni publisher non ha la necessità di conoscere ogni subscriber (e viceversa), ognuno di essi comunica esclusivamente con il broker.
- **Disaccoppiamento temporale**: un publisher e un subscriber non hanno la necessità di essere attivi simultaneamente per poter comunicare.
- **Disaccoppiamento di sincronizzazione**: le operazioni di invio e ricezione di messaggi non sono processi che interrompono il flusso di esecuzione.

Tipicamente MQTT instaura un canale di comunicazione bidirezionale sfruttando il protocollo *TCP/IP*, il quale garantisce affidabilità e robustezza del canale, ma soprattutto l'ordine dei messaggi inviati rimane invariato in ricezione.

Capitolo 3

Analisi

3.1 Obiettivi

Questa tesi si pone come obiettivo la realizzazione di un'applicazione web che permetta di trasformare il sistema emulato, utilizzato nella Notte Europea dei Ricercatori, in una scena in realtà aumentata sfruttando un dispositivo dotato di una fotocamera. L'applicativo deve inizialmente calibrare un'arena virtuale, sfruttando la telecamera del dispositivo e quattro marker ArUco posizionati nel mondo reale. Una volta individuati e calcolata una stima della loro posizione all'interno di una scena 3D, devono essere disegnati i robot emulati dal sistema, all'interno dell'arena virtuale. Le informazioni relative alla posizione ed orientamento di ogni elemento vengono condivise mediante un broker MQTT in un topic specifico.

3.2 Requisiti

- Ricerca dei marker ArUco presenti nell'inquadratura della telecamera fisica.
- Creazione di un arena virtuale in una scena 3D, sovrapposta ai marker posizionati nel mondo reale.
  - Calibrazione di un nuovo sistema di coordinate.
  - Posizionamento di una mesh sul piano dell'arena.

- Rappresentazione dei robot emulati nella scena in realtà aumentata ricevendo le informazioni dal broker MQTT.

3.3 Modello

In questo progetto viene implementato un sistema di riconoscimento di marker ArUco con l'ausilio di una telecamera. Il tutto integrato in un contesto di realtà aumentata, fornendo una rappresentazione più immersiva dell'emulatore. L'architettura del sistema viene illustrata in Figura 3.1, segue in Figura 3.2 il diagramma delle classi generale.

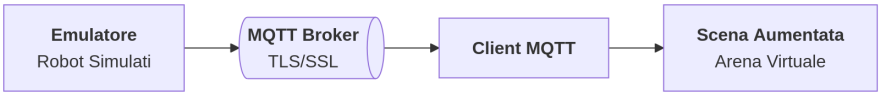


Figura 3.1: Architettura del sistema

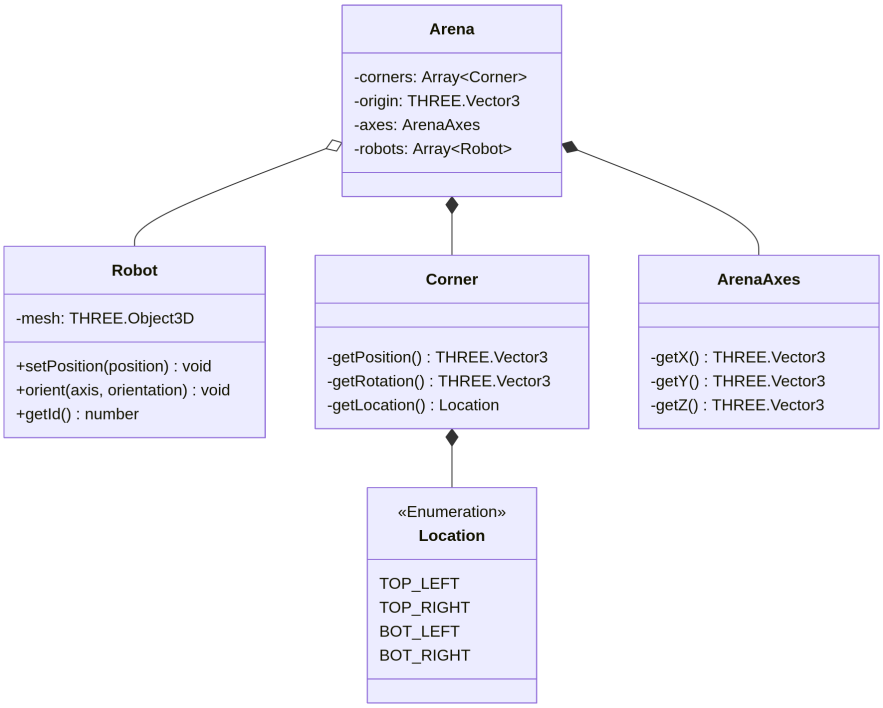


Figura 3.2: Diagramma delle classi del sistema

Nello specifico l'utente, nel momento in cui inizializza la sessione aumentata, approvando i permessi necessari, ha la possibilità di utilizzare la fotocamera esterna del dispositivo con lo scopo di inquadrare esattamente quattro marker ArUco. Una volta rilevati i marker viene realizzata, ed eventualmente calibrata, un'arena virtuale dove verranno rappresentati, nella scena 3D, i robot emulati dal sistema robot-emulation. I due sistemi comunicano attraverso MQTT, con l'obiettivo di condividere le informazioni che riguardano i robot simulati dall'emulatore con l'arena virtuale, per poter renderizzare i robot in realtà aumentata. Durante la simulazione è possibile muoversi liberamente all'interno della scena virtuale spostando il dispositivo di cattura utilizzato inoltre, in caso di disallineamento, l'utente in qualsiasi momento può ricalibrare l'arena virtuale.

3.3.1 Client MQTT

La classe `MQTTClient` incapsula i parametri da utilizzare quando viene effettuata la connessione ad un broker. In aggiunta fornisce dei metodi per inizializzare il processo di connessione e offre la possibilità di sottoscrivere il client a diversi topic. È stata progettata in maniera tale da poter essere riutilizzabile in atri contesti, fornendo meccanismi specifici che permettono di definire comportamenti diversi, ad esempio è possibile definire operazioni per gestire la disconnessione dal client, oppure i comportamenti da adottare in caso di ricezione dei messaggi.

3.3.2 Location

La `Location` definisce la posizione spaziale di ogni marker: i marker reali rappresentano un angolo dell'arena specifico, ognuno di essi ha un identificativo univoco utilizzato per distinguersi fra loro. In questo contesto è fondamentale distinguere i diversi marker in quanto, durante la fase di calibrazione, ognuno verrà usato in maniera differente, per ottenere dei risultati attendibili e più vicini alla realtà.

3.3.3 Corner

Un `Corner` (Figura 3.3) rappresenta un angolo dell'arena, al suo interno vengono incapsulati i dati relativi alla posizione, rotazione e `Location` del marker rilevato.

La sua rotazione dipende dall'inquadratura e dalla proiezione della telecamera, in quanto le immagini di una generica telecamera non sono ortogonali ma prospettiche. Ciò significa che ogni elemento, all'interno di un'immagine scattata da una telecamera, sarà distorto di un certo parametro definito dalla telecamera stessa. Mentre la sua `Location` viene definita a priori mediante l'identificativo del marker.

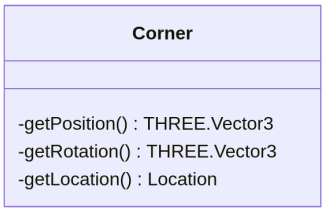


Figura 3.3: Diagramma delle classi di `Corner`

3.3.4 Arena Axes

La classe `ArenaAxes` racchiude i vettori normalizzati relativi agli assi calibrati dell'arena. Vengono calcolati in fasi di calibrazione sfruttando i `Corner` come punti di riferimento.
L'asse orizzontale punta verso il lato destro dell'arena, mentre quello verticale verso l'alto. Per quanto riguarda l'asse z viene considerato come vettore normale dell'arena durante la fase di posizionamento dei robot sul piano virtuale generato.

3.3.5 Robot

La classe `Robot` (Figura 3.4) racchiude le informazioni di un robot simulato all'interno dell'arena virtuale. Fornisce dei metodi per poter manipolare la rotazione e la traslazione in Sistema di Coordinate del Mondo (WCS), all'interno della scena 3D di ThreeJS. Vengono utilizzati gli `id` univoci, ricevuti dall'emulatore, per distinguerli fra loro nel momento in cui devono essere traslati o orientati. Inoltre utilizzano la stessa mesh con la stessa dimensione, ma rotazione e traslazione differente. È necessario conoscere la struttura della mesh, in quanto successivamente verranno utilizzati dei vettori normalizzati per effettuare il posizionamento corretto all'interno dell'arena. Per posizionamento si intende l'allineamento corretto sul piano virtuale, per simulare quelli reali posti su un piano.

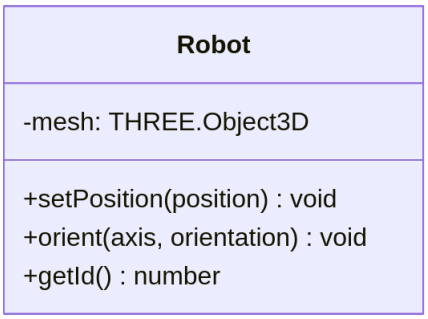


Figura 3.4: Diagramma delle classi di `Robot`

3.3.6 Arena

L'arena simulata (Figura 3.5) riproduce il comportamento dei robot all'interno del mondo reale, comunicando con il sistema principale mediante un broker MQTT. Questa simulazione pubblica tutte le informazioni relative alle posizioni ed orientamento all'interno di un topic. La soluzione implementata si sottoscrive ai relativi

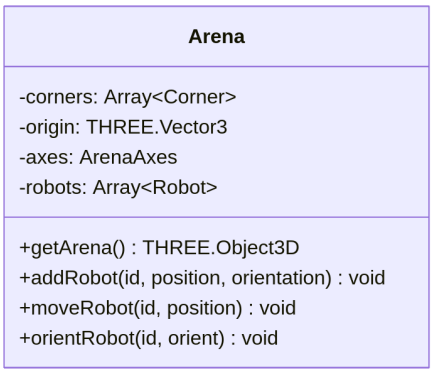


Figura 3.5: Diagramma delle classi dell'`Arena`

topic e al momento della ricezione di un messaggio, elabora i dati al suo interno per poi renderizzare i `Robot` all'interno della scena 3D. Uno dei problemi principali analizzati riguarda la differenza di sistema di coordinate di entrambe le arene: l'arena simulata non corrisponde all'arena virtuale nella scena, quindi una posizione nel sistema simulato, se utilizzata all'interno della scena senza alcuna manipolazione, comporta a posizionamenti errati all'interno della scena. Il posizionamento dei robot deve essere effettuato all'interno di un'area precisamente delimitata nella

scena, nasce la necessità di adattare le posizioni relative ricevute nel sistema di coordinate corrente. Oltre a questa trasformazione, si aggrega la trasformazione dalle coordinate relative all'arena al WCS.

Capitolo 4

Progettazione

4.1 Marker Detection e Pose Estimation

Nel flusso esecutivo il riconoscimento, essendo una parte molto onerosa, non viene eseguito ad ogni iterazione del *loop* principale, ma viene eseguita soltanto dopo un determinato numero di *frame* dall'ultima esecuzione se l'arena non è stata già calibrata. Ciò garantisce la fluidità dell'applicativo in fase di calibrazione, ma porta a non avere un tracking immediato dei marker presenti. Viene utilizzata una classe di supporto, chiamata `Marker` mostrata in Figura 4.1, per tenere traccia di tutti i marker trovati e le loro informazioni relative alla `pose`. La `pose` è un oggetto, generato dalla libreria *posit2* integrata con *js-aruco2* [Fal], contenente i seguenti parametri:

- **error**: un valore relativo all'errore stimato del risultato;
- **translation**: un vettore di traslazione 3D;
- **rotation**: una matrice di rotazione, di dimensione 3×3;

`Posit`, quando utilizzata, genera due possibili `pose`: una chiamata `bestPose`, avente il valore dell'errore inferiore, ed una seconda chiamata `alternativePose`, avente un errore maggiore. La classe `Marker` è stata predisposta per fornire entrambe le possibili `pose`; tuttavia nella soluzione implementata verrà utilizzata soltanto quella migliore.

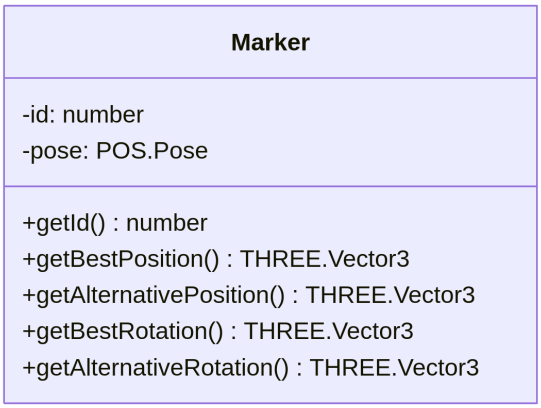


Figura 4.1: Diagramma delle classi di `Marker`

4.2 Arena virtuale

L'arena è delimitata da marker ArUco posizionati nel mondo reale e ha lo scopo di disegnare all'interno tutti i robot emulati dal simulatore esterno. Per creare e calibrare un'arena virtuale, è necessario definire quali sono i limiti dell'arena stessa. Questi limiti sono i `Corner`, la cui posizione nella scena virtuale viene definita dai marker ArUco ricercati nella fase precedente.
Ogni `Corner` necessita dei seguenti parametri:

- `position`: la posizione nella scena 3D;
- `rotation`: la rotazione nella scena 3D, influenzata dalla proiezione prospettica della fotocamera;
- `location`: indica quale angolo dell'arena rappresenta nel mondo reale (per esempio l'angolo in alto a destra, oppure in basso a sinistra);

Un'arena per essere consistente, deve presentare quattro `Corner` ognuno dei quali rappresenta una `Location` specifica. Le `Location` sono indispensabili per il calcolo degli assi relativi dell'arena stessa. Questi ultimi vengono utilizzati per posizionare i robot all'interno a partire dall'origine dell'arena.

4.2.1 Calibrazione dell'arena

Una volta aver ottenuto un'arena valida, bisogna effettuare delle opportune operazioni per poter permettere il posizionamento di robot all'interno dell'area delimitata (Figura 4.2). In particolare, il simulatore invia le posizioni dei robot relative all'origine dell'ambiente simulato $(0, 0, 0)$, tuttavia la posizione dell'origine dell'arena raramente corrisponde a quella del sistema emulato. Perciò le posizioni ricevute, relative all'origine della scena emulata, devono essere trasformate in posizioni relative al nuovo sistema di coordinate. Viene considerato il punto centrale

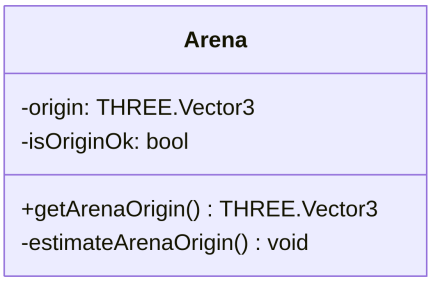


Figura 4.2: Diagramma delle classi parziale: `Arena` e l'origine

del poligono definito dai quattro angoli dell'arena, come origine del nuovo sistema di coordinate. In matematica, quest'ultimo è chiamato centroide (centro geometrico o centro della figura): si ottiene calcolando la media aritmetica della posizioni dei punti di una figura piana o solida.

$$C = \frac{1}{n} \sum_{i=0}^{n} x_i$$

Il punto trovato corrisponde all'origine dell'arena nel sistema di coordinate del mondo della scena 3D.

Calcolo degli assi di riferimento

Per permettere la traslazione dei robot nel nuovo sistema di riferimento, bisogna definire un'asse orizzontale ed uno verticale, entrambi relativi all'arena. Qui entra in gioco la `Location` di ogni `Corner`: data la posizione di due `Corner`, il vettore

normalizzato della differenza fra le posizioni definisce un asse dell'arena.
Per l'asse verticale si prendono in considerazione due angoli che si trovano entrambi a destra o a sinistra (ad esempio `TOP_LEFT` e `BOT_LEFT`). Il vettore risultante avrà il verso che punta all'angolo superiore. Mentre per l'asse orizzontale si prendono in considerazione due *corner* che si trovano, idealmente, sulla parte inferiore oppure sulla parte superiore (ad esempio `TOP_LEFT` e `TOP_RIGHT`).

4.2.2 Comunicazione con il broker MQTT

La comunicazione con un broker MQTT deve avvenire utilizzando una connessione sicura, in quanto il contesto dell'applicativo è sviluppato con il protocollo HTTPS. È essenziale che la comunicazione con il broker avvenga attraverso un canale sicuro, come ad esempio *TLS/SSL*, al fine di proteggere le informazioni da intercettazioni o manomissioni e garantire l'integrità dei dati scambiati. Soltanto in un ambiente sicuro, il browser potrà permettere la comunicazione con quest'ultimo.
Viene implementata la classe `MQTTClient` (Figura 4.3), che permette di connettersi ad un broker, sottoscriversi ai topic, ed eseguire delle operazioni specifiche quando un messaggio viene ricevuto.

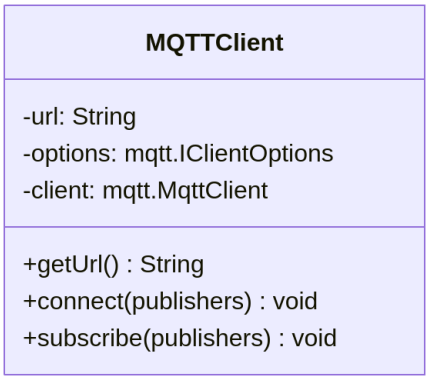


Figura 4.3: Diagramma delle classi di `MQTTClient`

Gestione dei Robot emulati

Le informazioni ricevute dal broker, in formato *json*, vengono memorizzate all'interno della classe `Robot`, con la struttura mostrata in Figura 4.4. Siccome tutti

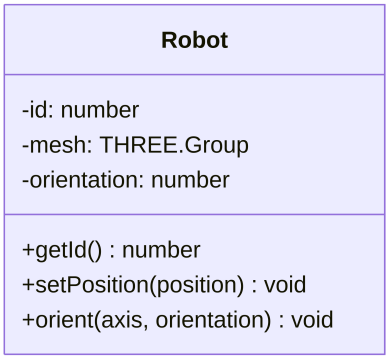


Figura 4.4: Diagramma delle classi del `Robot`

i robot utilizzano lo stesso modello di base, esso viene caricato una volta sola e, ogni volta che viene aggiunto un robot, viene creata una nuova copia con un nuovo identificativo nella scena di ThreeJS. Per ogni robot non è necessario tenere traccia delle posizioni relative all'emulatore: infatti ogni posizione viene convertita nel nuovo sistema di coordinate prima di creare un'istanza di questa classe.

4.2.3 Posizionamento e orientamento dei robot

Il posizionamento dell'arena virtuale non è sempre lo stesso, perciò non è possibile definire un posizionamento statico dei `Robot`. A causa del posizionamento dinamico si ha la necessità di allineare arbitrariamente la mesh. In particolare la base della mesh, contenente le ruote dei robot, deve essere posizionata sul piano virtuale dell'arena. Geometricamente il vettore normale della base deve avere stessa direzione ma verso opposto rispetto al vettore normale del piano dell'arena. In questa soluzione viene proposto un orientamento sfruttando le matrici di trasformazione della mesh e quella dell'arena: data la matrice di trasformazione iniziale M_i, si può dire che moltiplicando questa matrice per una matrice di rotazione R, si ottiene la matrice di trasformazione finale M_f.

$$M_f = M_i \cdot R$$

In questo caso la matrice M_i rappresenta la matrice di trasformazione della mesh, mentre la matrice M_f rappresenta quella dell'arena. Trovando la matrice R è

possibile applicare le opportune trasformazioni per orientare correttamente una mesh sul piano dell'arena.

$$R = M_f/M_i = M_f \cdot M_i^{-1}$$

La matrice dell'arena viene ricavata utilizzando gli assi calcolati dopo la calibrazione, invece la matrice dei `Robot` viene definita in maniera statica. Analizzando la mesh del robot, si determinano i vettori delle normali nello spazio di coordinate del mondo. In particolare bisogna definire il vettore normale della base, della parte frontale e del lato destro della mesh (destro perché viene utilizzato un sistema di coordinate che segue le regole della mano destra).
Per quanto riguarda l'orientamento dei robot, viene utilizzato l'asse y dell'arena come asse di rotazione, in modo da rispettare quello del simulatore. La rotazione, ogni volta che deve essere aggiornata, viene sommata o sottratta a quella corrente.

4.2.4 Movimento dei robot

Per poter muovere i robot all'interno dell'arena, bisogna calcolare la loro posizione all'interno della scena. Le posizioni ricevute dal broker MQTT vengono normalizzate, per cambiare sistema di riferimento nel sistema di coordinate relativo all'arena virtuale. Tuttavia queste coordinate normalizzate sono ancora relative ad un origine diversa da quella della scena, dunque non è possibile utilizzarle nel WCS. La classe `Arena` (Figura 4.5) fornisce un metodo che converte le coordi-

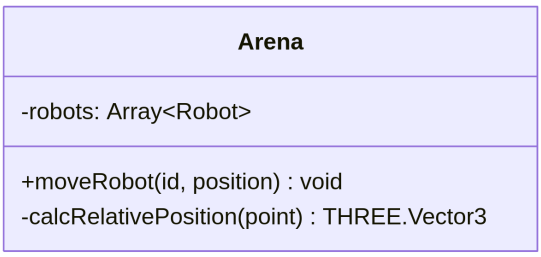


Figura 4.5: Diagramma delle classi parziale: `Arena` e movimento dei `Robot`

nate relative date in coordinate del mondo, permettendo una corretta traslazione delle mesh. Questo metodo sfrutterà sia l'origine che gli assi dell'arena: l'origine

definisce il punto in cui tutte le traslazione devono iniziare, mentre gli assi indicano le direzioni su cui devono essere effettuate. Non è sufficiente sommare le coordinate di una posizione relativa alle coordinate dell'origine, in quanto tutte le traslazioni avvengo nel WCS: supponendo di effettuare una traslazione lungo l'asse x del mondo, non si ha alcuna certezza che corrisponda a quella corretta nel piano dell'arena. In questo caso gli assi del mondo e dell'arena non corrispondono; utilizzando direttamente gli assi dell'arena il problema non si presenta in alcuna situazione.

Capitolo 5

Implementazione

Il flusso di esecuzione dell'applicazione è il seguente:

1. Inizializzazione della scena 3D e del supporto con WebXR, stabilisce una connessione con il broker MQTT ed inizializza il detector per i marker ArUco con l'opportuno dizionario

2. Nel momento in cui l'utente inizia la sessione in realtà aumentata, attraverso l'apposito pulsante, viene definito il loop di esecuzione con le seguenti fasi:

   - `update`, esegue la ricerca dei marker ed eventualmente inizializza un'arena virtuale;
   - `render`, disegna l'intera scena sullo schermo.

5.1 ThreeJS e WebXR

La realizzazione di una scena in realtà aumentata con l'ausilio di ThreeJS è molto immediato, in quanto la libreria integra la maggior parte delle funzionalità di WebXR. Le due librerie coesistono elaborando i dati in contemporanea: ThreeJS si occupa di renderizzare tutti gli elementi della scena e l'immagine della fotocamera ad ogni frame, mentre WebXR gestisce il dispositivo XR utilizzato ed eventuali controller.
Tuttavia, WebXR categorizza l'immagine delle telecamera, ed altre informazioni non rilevanti in questa tesi, come contenuto sensibile. Perciò per poter avviare

una qualsiasi sessione in realtà aumentata (o virtuale), è necessario che l'utente dia il consenso all'utilizzo della telecamera. Inoltre per ottenere l'immagine stessa durante una sessione XR, utilizzando il modulo aggiuntivo *WebXR Raw Camera Access Module* [Imm24]. Grazie a questo modulo si ha la possibilità di ottenere una texture, nel contesto di WebGL, rappresentante l'immagine della fotocamera in uno specifico `XRFrame`, durante l'esecuzione di una `XRSession`.

5.1.1 Integrazione di una telecamera

Per integrare la telecamera del dispositivo all'interno di una scena ThreeJS, è necessario aver inizializzato il renderer, successivamente bisogna abilitare il componente `xr` all'interno del renderer stesso.

Listato 5.1: Inizializzazione della scena 3D con il supporto per WebXR

```
// xr session features
const reqFeats = ["viewer", "camera-access"];
...
async function init() {
    ...
    // initialize WebGLRender with custom parameters
    renderer = new THREE.WebGLRenderer({
        antialias: true,
        canvas: canvas,
        preserveDrawingBuffer: true });
    // enable webxr support
    renderer.xr.enabled = true;

    // add helper button to initialize an XRSession with the given features
    document.body.appendChild(ARButton.createButton(renderer, {
        requiredFeatures: reqFeats,
        optionalFeatures: ["dom-overlay"],
        domOverlay: { root: divUi }
    }));
    ...
}
```

Per poter ottenere l'immagine della fotocamera, è necessario specificare la *feature* `camera-access` (Listato 5.1), fornita dal WebXR Raw Camera Access Module.

5.2 Marker Detection e Pose Estimation

5.2.1 Ricerca dei marker

Per ricercare all'interno di un'immagine i marker ArUco viene utilizzata la libreria js-aruco2. Questa libreria analizza i dati di un immagine e attraverso algoritmi e tecniche di Visione Artificiale, trovando la posizione dei marker ArUco di un dizionario specifico, all'interno dell'immagine data. Inoltre offre delle tecniche per stimare la posizione dei marker in uno spazio tridimensionale.
Fornisce i metodi necessari per effettuare la ricerca di marker ArUco appartenenti ad un dizionario specifico, partendo da un'immagine (solitamente ottenuta da un *canvas*).

Listato 5.2: Richiesta dell'immagine dalla fotocamera

```
const DETECT_FRAMES = 100;

function update(time) {
    if (renderer.info.render.frame % DETECT_FRAMES == 0 && !calibrated) {
        const image = getCameraImage();
        if (!image) return;
    }
    ...
}

function getCameraImage() {
    // it is possible to retrieve camera parameters only
    // in an XRSession context
    if (!renderer.xr.getSession()) return;

    // current generated renderer frame
    const frame = renderer.xr.getFrame();
    // current reference space
    const refSpace = renderer.xr.getReferenceSpace();
    if (!frame || !refSpace)
        return;
    // retrieves the view pose
    const viewPose = frame.getViewerPose(refSpace);
    if (!viewPose)
        return;
    // retrieves the device used, in this case a basic webcam is supported
    const view = viewPose.views.find(view => view.camera);
    if (!view)
        return;
    // retrieve camera texture
```

```
31      const camText = renderer.xr.getCameraTexture(view.camera);
32      if (!camText)
33          return;
34
35      // draw the camera content in another scene as background
36      imageScene.background = camText;
37      imageScene.background.needsUpdate = true;
38
39      // creates an images by taking a screenshot of the renderer scene
40      // using the given camera
41      const imageData = Utils.snapshot(renderer, camera, imageScene);
42
43      return imageData;
44  }
```

In questo caso non è possibile ottenere l'immagine della fotocamera in maniera diretta. Grazie alla feature `camera-access`, è possibile ottenere una texture (Listato 5.2), contenente l'immagine della fotocamera, in un formato non elaborabile lato CPU (`WebGLTexture`). Tuttavia è possibile utilizzare questo specifico formato come sfondo di una scena. Per questo motivo, viene utilizzata una scena ausiliaria con lo scopo di renderizzare la texture ottenuta, per poi effettuare uno *snapshot* dello stato attuale della scena. Così facendo è possibile realizzare un immagine elaborabile da js-aruco2. Prima di utilizzarla bisogna capovolgere l'immagine: il metodo `readRenderTargetPixels`, all'interno del (Listato 5.3), richiama il metodo di *OpenGL* chiamato `glReadPixels`. Quest'ultimo inizia a leggere i pixel dall'angolo in basso a sinistra dell'immagine, mentre l'origine della finestra è posta all'angolo in alto a sinistra. Questa differenza nelle origini porta ad avere un'immagine capovolta verticalmente. Il metodo `flipImageVertically` analizza metà dell'immagine, partendo dall'alto, e ad ogni iterazione scambia la riga corrente con la rispettiva riga dal fondo.

Listato 5.3: Snapshot della scena di supporto per ottenere un immagine elaborabile

```
1   function snapshot(renderer, camera, scene) {
2       const width = renderer.domElement.width;
3       const height = renderer.domElement.height;
4       // create render target
5       const rt = new THREE.WebGLRenderTarget(width, height);
6
7       // retrieve current render target
8       const oldRt = renderer.getRenderTarget();
9       renderer.setRenderTarget(rt);
10      renderer.render(scene, camera);
```

```
11
12      // create a buffer to store render target data
13      // 4 bytes of data => RGBA texture format
14      const buffer = new Uint8Array(width * height * 4);
15      renderer.readRenderTargetPixels(rt, 0, 0, width, height, buffer);
16      // reset old render target
17      renderer.setRenderTarget(oldRt);
18
19      // flip image vertically
20      return flipImageVertically(new ImageData(
21          new Uint8ClampedArray(buffer),
22          width,
23          height
24      ));
25  }
26
27  function flipImageVertically(image) {
28      const { width, height, data } = image;
29      const rowSize = width * 4;
30      const temp = new Uint8ClampedArray(rowSize);
31      for (let y = 0; y < Math.floor(height / 2); y++) {
32          const top = y * rowSize;
33          const bottom = (height - y - 1) * rowSize;
34          // fill temp with top row data
35          temp.set(data.subarray(top, top + rowSize));
36          // copy in top row the bottom row data
37          data.copyWithin(top, bottom, bottom + rowSize);
38          // set bottom row with top data
39          data.set(temp, bottom);
40      }
41      return image;
42  }
```

Una volta ottenuta l'immagine viene processata da js-aruco2, ottenendo le posizioni dei marker in coordinate 2D.

Listato 5.4: Ricerca dei marker ArUco

```
1   // aruco detector
2   const detector = new AR.Detector({
3       dictionaryName: "ARUCO"
4   });
5
6   function update(time) {
7       if (renderer.info.render.frame % DETECT_FRAMES == 0 && !calibrated) {
8           const image = getCameraImage();
9           if (!image) return;
10          const markers = detectMarkers(image);
11      }
12      ...
```

```
}

function detectMarkers(imageData) {
    // detect all marker screen coordinates
    return detector.detect(imageData);
}
```

Dal Listato 5.4 si nota che il riconoscimento, come l'acquisizione dell'immagine, viene effettuato soltanto dopo l'esecuzione di un preciso numero di frame. Questo perché, come viene spiegato nel capitolo successivo, le fasi di riconoscimento e di stima delle posizioni, sono molto dispendiose a livello di risorse. Per questo motivo non viene implementato un riconoscimento in tempo reale. Tuttavia, per fornire maggiore flessibilità, viene fornita all'utente la possibilità di ricalibrare l'arena in qualsiasi momento, mediante l'utilizzo di un'interfaccia grafica dedicata. Quando la ricalibrazione viene effettuata, l'arena virtuale precedente viene temporaneamente mantenuta attiva, fino al momento in cui vengono tracciati i nuovi marker. Segue la ricreazione dell'arena e l'acquisizione delle informazioni iniziali ed alla ricalibrazione del sistema.

5.2.2 Stima delle posizioni

La stima delle posizioni viene calcolata sfruttando la libreria posit2, fornita da js-aruco2. La quale all'interno calibra una telecamera virtuale, conoscendo la larghezza dell'elemento *HTML* dove viene renderizzata la scena e la grandezza dei marker ArUco reali, per poter trasformare le coordinate nello spazio dello schermo in coordinate della scena. Per poter ottenere dei risultati attendibili, è necessario centrare le coordinate 2D dei corner, trovati nella sezione 5.2.1, all'interno del canvas (Listato 5.5).

Listato 5.5: Stima delle posizioni dei marker in uno spazio 3D

```
function update(time) {
    if (renderer.info.render.frame % DETECT_FRAMES == 0 && !calibrated) {
        ...
        const markers = detectMarkers(image);
        trackMarkers(markers);
    }
    ...
}

```

```
function trackMarkers(markers) {
    if (markers.length == 0) return;

    tracked = [];
    // initialize posit object to estimante 3D position
    const posit = new POS.Posit(modelSize, renderer.domElement.width);
    markers.forEach(m => {
        let corners = m.corners;

        // center all corners in the canvas
        for (let i = 0; i < corners.length; ++i) {
            let corner = corners[i];
            corner.x = corner.x - (renderer.domElement.width / 2);
            corner.y = (renderer.domElement.height / 2) - corner.y;
        }

        // estimate the marker rotation and traslation
        const pose = posit.pose(corners);
        const t = new Marker(m.id, pose);

        // exclude already tracked markers
        if (!tracked.find(e => e.getId() == t.getId())) {
            tracked.push(t);
        }
    });
    calibrated = tracked.length === 4;
    ...
}
```

Vengono tracciati tutti i marker, con `Id` unico, che successivamente, dopo le opportune manipolazioni, verranno utilizzati per la creazione dell'arena virtuale.

5.3 Arena Virtuale

5.3.1 Calibrazione dell'arena

L'inizializzazione dell'arena avviene nel momento in cui sono stati tracciati esattamente quattro marker differenti. A questo punto si procede con il calcolo dell'origine e degli assi di traslazione dell'arena.

Calcolo dell'origine

Per calcolare il centroide di una figura ad n dimensioni, bisogna calcolare la media di ogni coordinata per ogni suo punto (Listato 5.6). In questo caso si hanno 3 dimensioni, quindi si calcola la media delle coordinate x, la media delle coordinate y e la media delle coordinate z di ogni punto. Le medie ottenute rappresentano le coordinate tridimensionali del centroide della figura.

Listato 5.6: Calcolo del centroide di una serie di punti

```
class Arena {
    ...
    #estimateArenaOrigin() {
        const x = this.#calculateCentroid(this.#corners.map(p => p.getPosition().x
            ));
        const y = this.#calculateCentroid(this.#corners.map(p => p.getPosition().y
            ));
        const z = this.#calculateCentroid(this.#corners.map(p => p.getPosition().z
            ));

        this.#origin = new THREE.Vector3(x, y, z);
        this.#isOriginOk = true;

        this.#arena.position.set(this.#origin.x, this.#origin.y, this.#origin.z);
    }

    #calculateCentroid(points) {
        if (points === undefined || points.length === 0) {
            console.error("[ERROR] points cannot be empty");
            return undefined;
        }
        if (points.length == 1) return points[0];
        return points.reduce((acc, val, _) => acc + val) / points.length;
    }
    ...
}
```

Definizione degli assi

Conoscendo gli `Id` dei marker tracciati, ognuno dei quali corrisponde ad un angolo della scena specifico (`Location`), è possibile ottenere gli assi di traslazione dell'arena creata (Listato 5.7).

Listato 5.7: Calcolo degli assi di traslazione dell'arena

```
class Arena {
    ...
    #getCornerFromLocation(location) {
        if (!Location.isValid(location)) return undefined;
        return this.#corners.find(c => c.getLocation() == location);
    }

    #estimateArenaAxes() {
        const topLeft = this.#getCornerFromLocation(Location.TOP_LEFT);
        const topRight = this.#getCornerFromLocation(Location.TOP_RIGHT);
        const xaxis = new THREE.Vector3().subVectors(topRight.getPosition(),
            topLeft.getPosition()).normalize();

        const botLeft = this.#getCornerFromLocation(Location.BOT_LEFT);
        const yaxis = new THREE.Vector3().subVectors(topLeft.getPosition(),
            botLeft.getPosition()).normalize();

        this.#axes = new ArenaAxes(xaxis, yaxis, new THREE.Vector3().crossVectors(
            xaxis, yaxis));
        this.#isAxesOk = true;
    }
    ...
}
```

Si prendono due angoli, che nella realtà si troverebbero sullo stesso lato, e si calcola il vettore distanza fra loro:

$$dist = p_2 - p_1$$

Il risultato ottenuto viene poi normalizzato, così da ottenere valori compresi nell'intervallo $[-1, 1]$ per facilitare le operazioni di traslazione e rotazione. Inoltre viene anche calcolato un terzo asse, calcolando il prodotto vettoriale tra i due assi calcolati in precedenza.

$$z = x \times y$$

Quest'ultimo asse può essere considerato come il vettore normale della superficie dell'arena; in seguito verrà utilizzato per orientare correttamente la mesh dei robot sulla superficie stessa.

5.3.2 Comunicazione con il broker

Nel momento in cui l'arena viene creata, viene definito il comportamento di un `MQTTClient` alla ricezione di ogni messaggio.

Listato 5.8: Creazione del client MQTT ed iscrizione ai topic

```
const url = "wss://mqtt-broker-hostname:9001";
const topics = [
    "robots/+/position"
]
const opts = {
    protocol: "wss",
    clean: true,
    connectTimeout: 4000,
    reconnectPeriod: 2000,
    rejectUnauthorized: true
}

// create an mqtt client with the given options
const client = new MQTTClient(url, opts);
// try to connect to the broker
// when it succeed it subscribes to the given topic list
client.connect(topics);
```

In questo caso l'unico topic rilevante è quello delle posizioni dei robot (`robots/{robotId}/position`), come mostrato nel Listato 5.8, perciò non è necessario filtrare i messaggi per topic (essendo iscritto soltanto ad uno). Ogni volta che viene ricevuto un messaggio, viene normalizzata la posizione ricevuta (Listato 5.9), nelle coordinate dell'arena virtuale: come specificato in precedenza, l'arena simulata e l'arena virtuale non solo hanno due sistemi di coordinate differenti, ma anche la dimensione delle due arene è differente. Per questo motivo tutte le posizioni ricevute dal simulatore, vengono normalizzate con la seguente formula:

$$normPos = \left(\frac{x \cdot xSize}{simulatedSize}, \frac{y \cdot ySize}{simulatedSize}, z \right)$$

Dove $xSize$ e $ySize$ rappresentano rispettivamente la metà della lunghezza di due lati consecutivi dell'arena, mentre $simulatedSize$ è la dimensione del mondo dell'arena simulata.

Listato 5.9: Operazioni eseguite alla ricezione di ogni messaggio

```
async function createArena(bestValues = true) {
    ...
    // clear arena corners if already exist
    if (arena) {
        // remove arena system from the scene
        const arenaObj = arena.getArena();
        scene.remove(arenaObj);
```

```
8           // clear arena corners
9           arena.clearCorners();
10      }
11
12      arena = new Arena(corners, simWorldSize);
13      arena.createCasters();
14      // add the arena 3d virtual object to the scene
15      scene.add(arena.getArena());
16
17      client.on('message', async (topic, msg) => {
18          // json parsed content of mqtt message
19          const json = parseBrokerMessage(msg);
20          const rId = json.robot_id;
21          const simPos = { x: json.x, y: json.y };
22          const orient = json.orientation;
23          // convert simulated arena coords into arena coords
24          const normPos = Arena.normalizeSimulatedPos(arena, simPos);
25
26          // if the robot is not in the arena it will be created
27          if (!arena.hasRobot(rId)) {
28              await arena.addRobot(rId, normPos, orient);
29          }
30
31          // robot position
32          arena.moveRobot(rId, normPos)
33          // robot arena y-axis orientation
34          arena.orientRobot(rId, orient)
35      });
36  }
37
38  class Arena {
39      ...
40      static normalizeSimulatedPos(arena, position) {
41          const arenaSize = arena.getArenaSize();
42          const simSize = arena.getSimulatedSize();
43
44          const xfactor = arenaSize.x / simSize;
45          const yfactor = arenaSize.y / simSize;
46
47          return new THREE.Vector3(position.x * xfactor, position.y * yfactor, 0);
48      }
49  }
```

All'interno del listato 5.9 sopra mostrato, si nota come vengono definite le operazioni da eseguire quando viene ricevuto un messaggio: ogni volta che il client riceve un messaggio emette il segnale `message`, mentre nel programma principale vengono definite le istruzioni da eseguire ogni volta che il segnale viene emesso.

5.3.3 Posizionamento e orientamento

A questo punto viene calcolata la matrice di trasformazione per poter allineare la mesh con il piano dell'arena. Si calcolano le matrici di trasformazione sia iniziale, della mesh, che finale, corrispondente al piano dell'arena (Listato 5.10). Nel sistema di coordinate della mesh, il vettore normale del robot corrisponde all'asse delle y avente verso opposto, per questo motivo, nella matrice di trasformazione finale, `tb`, vengono scambiati l'asse z e l'asse y. Infatti la base della mesh deve posizionarsi nella stessa direzione della normale dell'arena (asse z). Inoltre lo stesso asse z viene negato per permettere alla base di posizionarsi correttamente. Calcolate le due matrici, si effettua un prodotto tra quella finale (di arrivo), e l'inversa di quella iniziale, così da ottenere la matrice di rotazione da applicare alla mesh.

Listato 5.10: Posizionamento della mesh sull'arena virtuale

```
1  // mesh base normal vector
2  const ROBOT_BASE = new THREE.Vector3(0, -1, 0);
3  // mesh normal vector used to rotate in local space
4  const ROBOT_UP = ROBOT_BASE.clone().negate();
5  // mesh front face normal vector
6  const ROBOT_FRONT = new THREE.Vector3(1, 0, 0);
7  // mesh right normal vector
8  const ROBOT_RIGHT = new THREE.Vector3(0, 0, 1);
9
10 class Arena {
11     async addRobot(id, position, orientation) {
12         ...
13         const robotPos = this.#calcRelativePosition(position);
14         const mesh = await createRobotMesh(robotPos);
15         // robot initial matrix
16         const rb = new THREE.Matrix4().makeBasis(ROBOT_RIGHT, ROBOT_BASE,
               ROBOT_FRONT);
17         // robot target matrix
18         const tb = new THREE.Matrix4().makeBasis(this.#axes.getX(), this.#axes.
               getZ().negate(), this.#axes.getY());
19         // mat * rb = tb => mat = tb / rb = tb * 1/rb = tb * inverse(rb)
20         const mat = new THREE.Matrix4().multiplyMatrices(tb, rb.clone().invert());
21         mesh.rotation.setFromRotationMatrix(mat);
22         // adjust mesh orientation
23         mesh.rotateOnAxis(ROBOT_UP, orientation);
24
25         // create new robot
26         const robot = new Robot(id, mesh, orientation);
27         this.#robots.push(robot);
28         ...
```

```
29      }
30  }
```

Per quanto riguarda l'orientamento dei `Robot`, illustrato nel Listato 5.11, bisogna tenere traccia dell'orientamento precedente per poter calcolare un *offset*, da utilizzare per effettuare la rotazione locale della mesh. Nel caso in cui si ruotasse la mesh reimpostando la rotazione, si perderebbe il posizionamento sull'arena, in quanto la matrice di rotazione calcolata precedentemente verrebbe sovrascritta da una nuova. Per questo motivo, dopo aver calcolato la differenza fra l'orientamento precedente e quello attuale, si utilizza il metodo `rotateOnAxis`, che permette di calcolare una rotazione rispetto ad un asse sommandola alla rotazione corrente di un `Object3D`. Infine si aggiorna l'orientamento attuale del `Robot` con il nuovo valore.

Listato 5.11: Rotazione del robot all'interno dell'arena

```
1   class Robot {
2       ...
3       orient(axis, orient) {
4           const offset = this.#orientation - orient;
5           this.#orientation = orient;
6           this.#mesh.rotateOnAxis(axis, offset);
7       }
8   }
9
10  class Arena {
11      ...
12      orientRobot(id, orient) {
13          const robot = this.#robots.find(r => r.getId() === id);
14          if (robot) {
15              robot.orient(ROBOT_UP, orient)
16          }
17      }
18  }
```

5.3.4 Movimento dei robot

Prima di effettuare una qualsiasi trasformazione si ricalcolano gli assi e l'origine dell'arena, se necessario: ciò accade quando vengono modificate le coordinate nel mondo di uno dei quattro `Corner` dell'arena (Listato 5.12). La posizione di

ogni robot viene calcolata partendo dall'origine dell'arena, a cui viene sommato il vettore corrispondente alla traslazione sull'asse x e y.

Listato 5.12: Traslazione dei robot all'interno dell'arena

```
class Robot {
    ...
    setPosition(position) {
        this.#mesh.position.copy(position);
    }
}

class Arena {
    ...
    moveRobot(id, position) {
        // find the given robot
        const robot = this.#robots.find(r => r.getId() === id);
        if (robot) {
            // update robot position if it exists
            robot.setPosition(this.#calcRelativePosition(position));
        }
    }

    #calcRelativePosition(point) {
        // calculates arena origin if needed
        if (!this.#isOriginOk) this.#estimateArenaOrigin();
        // calculates arena axes if needed
        if (!this.#isAxesOk) this.#estimateArenaAxes();

        // robot position start from arena origin point
        const robotPos = new THREE.Vector3().copy(this.#origin);
        // calculate final position using the given relative position
        const relx = new THREE.Vector3().copy(this.#axes.getX()).multiplyScalar(
            point.x)
        const rely = new THREE.Vector3().copy(this.#axes.getY()).multiplyScalar(
            point.y)
        return robotPos.add(relx).add(rely);
    }
}
```

Capitolo 6

Valutazione

In questo capitolo vengono discussi i risultati ed i limiti del sistema allo stato attuale, valutando il raggiungimento dei requisiti elencati nella Sezione 3.2. Sono state analizzate delle situazioni specifiche per ogni test affrontato, eventualmente aggiungendo ulteriori scene non necessariamente in realtà aumentata. Inoltre sono stati implementati script in Python per poter generare grafici e inviare messaggi specifici all'emulatore.

6.1 Stabilità del rilevamento dei marker

Il riconoscimento e il calcolo delle posizioni dei marker reali è un punto fondamentale per il corretto funzionamento dell'applicazione. I problemi che possono verificarsi sono numerosi: l'illuminazione può creare degli artefatti durante l'analisi dell'immagine portando ad un mancato tracciamento di un marker, come anche analizzando un immagine non perfettamente nitida. Inoltre la stima delle posizioni viene effettuata utilizzando POSIT, un algoritmo iterativo e approssimato della posizione reale, perciò valutando gli stessi dati iniziali più volte è possibile ottenere risultati differenti.
Questo fenomeno principalmente è causato da variazioni, anche minime, di numeri decimali durante le iterazioni dell'algoritmo. Per questo motivo il risultato ottenuto non è sempre preciso e uguale ad ogni avvio dell'applicativo.

Analisi dell'errore della stima

Per analizzare l'errore del calcolo delle posizioni dei marker rilevati, vengono utilizzati i seguenti elementi:

- un marker reale posizionato su un piano;
- una telecamera fissa;
- uno script in JavaScript capace di catturare diversi snapshot per poi stimare le posizioni usando POSIT;
- uno script in Python per calcolare la media e la varianza degli errori registrati e rappresentare i dati in un grafico

La telecamera inquadra il marker, fornendo l'immagine allo script JavaScript che effettua la ricerca dei marker, all'interno dell'immagine, per poi stimarne la posizione 3D; in seguito viene registrato il `bestError`, fornito da POSIT. Infine i dati ottenuti vengono analizzati dallo script in Python calcolando la media e la varianza per poi rappresentarli in un diagramma degli estremi, mostrato in Figura 6.1.

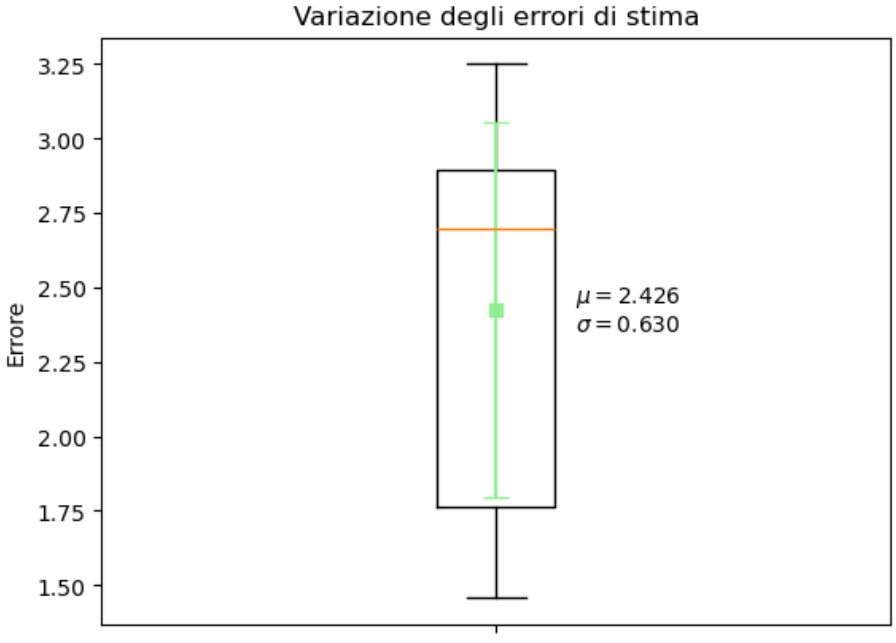


Figura 6.1: Variazione degli errori di POSIT

Un errore elevato comporta un forte disallineamento tra la posizione reale di un marker e quella stimata nella scena, un errore minore mostrerà risultati più accurati.

Prestazioni del riconoscimento

Per l'analisi delle prestazioni vengono utilizzati gli stessi strumenti utilizzati per l'analisi precedente nella sezione 6.1. La fase di riconoscimento di marker all'interno di un'immagine, stima delle posizioni compresa, è un'operazione molto onerosa, come mostrato in Figura 6.2. Segue la necessità di non utilizzare un

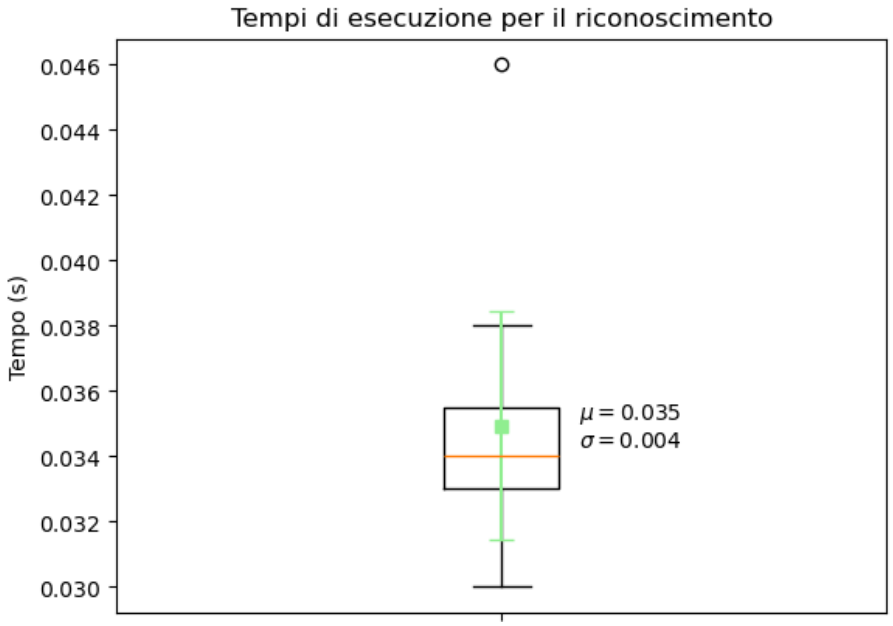


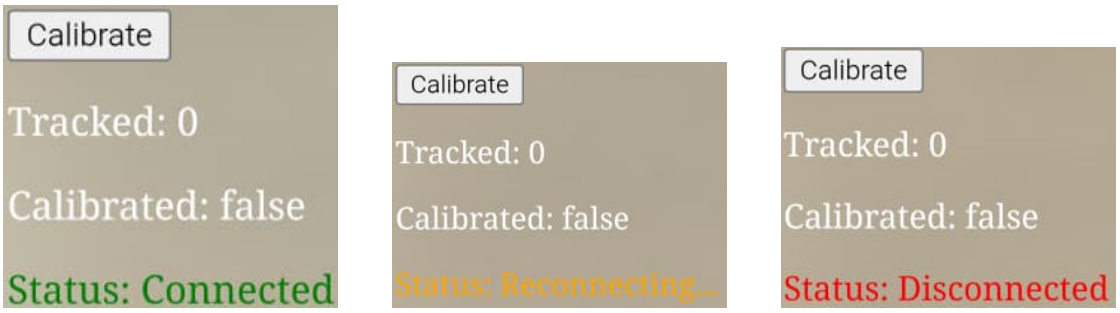
Figura 6.2: Tempi di esecuzione per ricercare i marker in un immagine

costante riconoscimento delle immagini ma, vengono limitate il numero di volte che viene eseguito al secondo, integrando un sistema di ricalibrazione manuale da poter effettuare in qualsiasi momento.

6.2 Gestione delle comunicazioni

La comunicazione con il broker viene impiegata esclusivamente per ottenere i dati relativi alla posizione di ogni robot emulato. Ad ogni messaggio ricevuto il siste-

ma assegna le nuove posizioni, convertite nel sistema di coordinate corrente, ed aggiunge eventuali robot se non sono presenti all'interno della scena. Nel caso in cui la connessione viene persa, quindi il client MQTT non riesce a comunicare con il broker, il sistema entra in fase di riconnessione. In questo periodo l'applicativo non viene interrotto ma ritenta la connessione al broker costantemente dopo un'intervallo di tempo specifico. Tuttavia l'arena, non ricevendo più messaggi, smette di aggiornare le posizioni dei robot ma non impedisce all'utente di muoversi liberamente all'interno. In questo modo viene garantito un funzionamento, anche se minimale, dell'applicativo. Inoltre, una volta che la connessione viene ristabilita, il sistema torna al suo normale funzionamento. Per avvisare l'utente in queste situazioni critiche, come mostrato nella Figura 6.3, viene implementata un'interfaccia grafica aggiuntiva che visualizza lo stato attuale della connessione: "connesso", "disconnesso", "riconnessione".



(a) Connesso (b) Riconnessione (c) Disconnesso

Figura 6.3: Stato della connessione con il broker

6.3 Gestione dell'arena virtuale

Il sistema dell'arena virtuale ha soddisfatto pienamente i requisiti proposti. Le fasi di calcolo dell'origine e degli assi sono accurate, garantendo un movimento realistico. Tuttavia l'immersività della scena, ovvero l'accuratezza del posizionamento dell'arena nel mondo aumentato, dipende dalla fase di riconoscimento e posizionamento dei marker all'interno della scena. Nella Figura 6.4 è possibile notare come la sovrapposizione degli angoli dell'arena e dei marker non sia perfetta, a causa dei fenomeni illustrati nella Sezione 6.1.

Nei test eseguiti i limiti dell'arena vengono rappresentati come cubi all'interno della scena, ai quali vengono aggiunti gli assi del sistema di coordinate locale per mostrare la rotazione dovuta alla proiezione della fotocamera. Le immagini sono state catturate tramite un dispositivo mobile dotato di fotocamera e sensori aggiuntivi, con lo scopo di fornire una misurazione dell'orientamento della fotocamera molto più accurato. Così facendo è possibile che si verifichino delle situazioni dove questi sensori creano disturbi, ad esempio quando il dispositivo è perfettamente orizzontale puntando il pavimento, portando ad un'inconsistenza nei dati alterando il risultato finale.

Figura 6.4: Esempio di arena virtuale delimitata dai marker

Allineamento della mesh L'allineamento della mesh viene eseguito in maniera dinamica, utilizzando sia la matrice di trasformazione statica di una mesh dei robot, che la matrice di trasformazione dinamica dell'arena virtuale. Come spiegato nella Sezione 4.2.3, si calcola la matrice di rotazione che permette di posizionarli sul piano dell'arena, mostrato in Figura 6.5. Vengono ottenuti dei risultati molto attendibili e sempre coerenti con la realtà, dunque è possibile affermare di aver soddisfatto il requisito dell'allineamento.

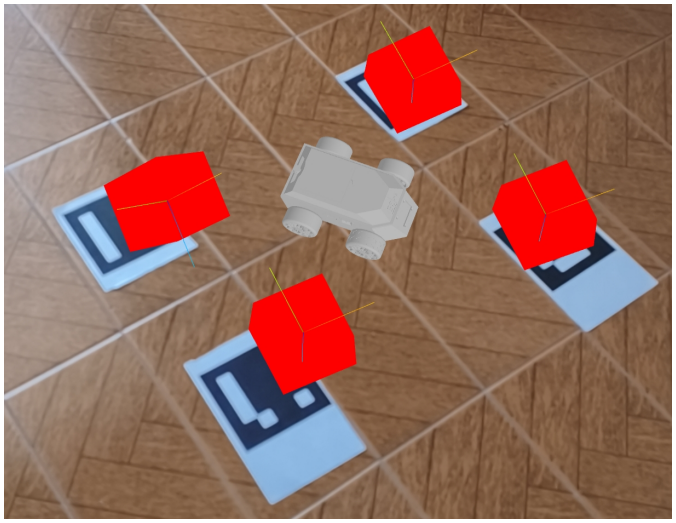
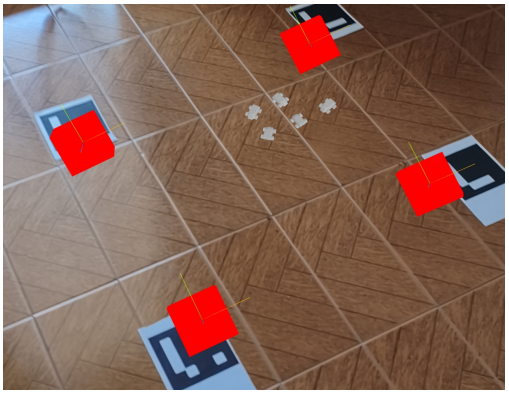


Figura 6.5: Dimostrazione dell'allineamento della mesh dei robot

Rappresentazione dei robot emulati Nelle figure sotto mostrate è possibile notare il posizionamento dei robot emulati. In particolare l'applicativo riceve correttamente le posizioni iniziali, situazione risultante in Figura 6.6a, ed istanzia correttamente il numero delle entità simulate, in questo caso specifico sono stati emulati esattamente 5 robot. Successivamente, in un'istante di tempo diverso,



(a) Posizionamento di alcuni robot all'interno dell'arena

(b) Movimento di tutti i robot all'interno dell'arena

uno script Python invia comandi specifici all'emulatore iniziando così a gestire il movimento dei robot, e ad aggiornare costantemente i topic MQTT. Il sistema, ad ogni messaggio ricevuto, aggiorna la posizione del robot virtuale, effettuando

le opportune trasformazioni delle coordinate, come spiegato nella Sezione 4.2.4 ed implementato nella Sezione 5.3.4. Dopo qualche secondo si ottiene il risultato illustrato in Figura 6.6b.

Da queste due ultime considerazioni si può affermare di aver raggiunto gli obiettivi prestabiliti all'inizio dello sviluppo del progetto, tuttavia con qualche limitazione. I diversi sistemi comunicano fra loro e possono coesistere nello stesso momento senza interferire l'uno con l'altro.

6.4 Stabilità del sistema

In questo paragrafo viene analizzata la stabilità dell'applicativo analizzando le prestazioni gestendo numerosi robot nello stesso momento. Vengono presi in considerazione i tempi impiegati sia per renderizzare tutti i robot, sia il tempo impiegato ad aggiornare tutte le posizioni. Sono stati implementati degli script in Python che permettono di inviare dei messaggi al broker, che verranno elaborati dall'emulatore per spostare i robot nel mondo emulato. In Figura 6.7 viene riportato

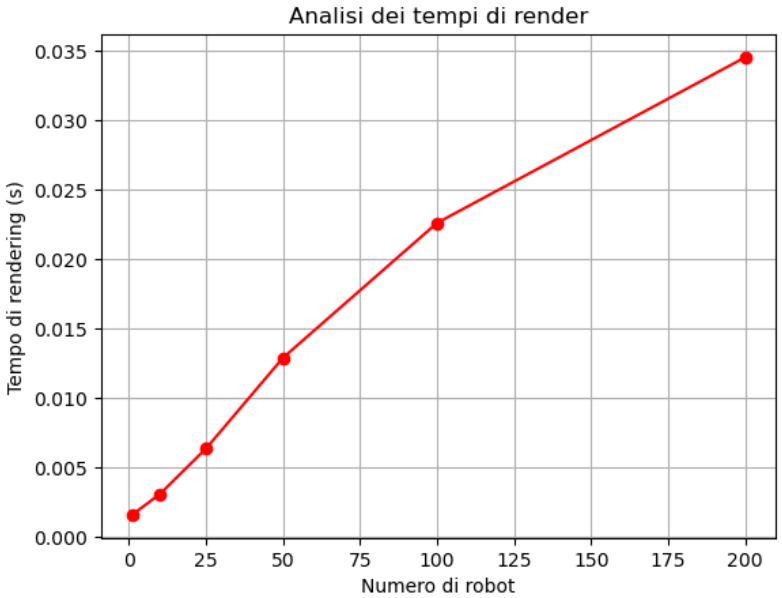


Figura 6.7: Grafico dei tempi di render della scena

il grafico che mostra l'analisi dei tempi di esecuzione durante la fase di rendering

in una scena statica. L'esecuzione rimane fluida fino al raggiungimento di circa 50 robot nello stesso momento. Il tempo impiegato non solo dipende dalla qualità della mesh utilizzata, ma soprattutto dal numero elevato di *draw call* che ThreeJS deve effettuare ogni frame.

Impatto della comunicazione

Per analizzare i tempi di ricezione ed elaborazione dei messaggi sono stati effettuati dei test con un numero elevato di robot simulati, nonostante ciò il tempo di esecuzione rimaneva molto basso, senza influenzare le prestazioni del sistema. Lo stesso vale durante la fase di movimento dei robot: siccome le operazioni da eseguire sono le stesse, queste non influenzano il normale funzionamento dell'applicativo.
Per analizzare i tempi di ricezione ed elaborazione dei messaggi sono stati effettuati dei test con un numero elevato di robot simulati, nonostante ciò il tempo di esecuzione rimaneva molto basso, senza influenzare le prestazioni del sistema. Lo stesso vale durante la fase di movimento dei robot: siccome le operazioni da eseguire sono le stesse, queste non influenzano il normale funzionamento dell'applicativo. Detto ciò è possibile concludere quest'analisi affermando che le prestazioni non sono influenzate dall'elaborazione dei messaggi ricevuti dall'emulatore, ma dipendono principalmente da due operazioni:

- l'analisi del contenuto di un'immagine per ricercare i marker e l'eventuale stima delle posizioni;
- il numero di mesh renderizzate all'interno della scena nello stesso istante.

Capitolo 7

Conclusioni e lavori futuri

In questo progetto è stato implementato correttamente un sistema capace di trasformare l'arena emulata in un contesto più immersivo tramite la realtà aumentata. Tutti i requisiti sono stati raggiunti con alcune limitazioni, quali:

- il rilevamento dei marker non in tempo reale a causa dell'impatto sulle prestazioni, influenzando la reattività della scena;
- i corner non sono perfettamente allineati con i marker reali, a causa degli errori durante la stima delle posizioni 3D;

Nonostante queste limitazioni l'applicativo riesce comunque a gestire un'arena virtuale che corrisponde all'arena simulata dall'emulatore. I robot vengono posizionati correttamente sul piano dell'arena, i numerosi calcoli fra matrici, molto dispendiosi, non influenzano la fluidità del sistema aumentato e la comunicazione non ha un impatto significativo su quest'ultimo. Tutti gli studi sono stati effettuati sulla macchina locale, nel caso in cui l'applicazione venga eseguita in remoto, ovvero broker e server web non vengono eseguiti sulla stessa macchina, le traslazioni potrebbero subire un disallineamento temporale, creando un movimento non più lineare.

Questo progetto, con le limitazioni attuali, potrebbe essere utilizzato come risorsa per un nuovo sistema completamente ristrutturato, oppure fornire una base di partenza per uno prossimo sviluppo. In questo caso, le possibili funzionalità critiche da integrare sono le seguenti:

- un'implementazione più robusta e stabile del rilevatore di marker AruCo assieme ad un sistema migliorato per stimarne la posizione in una scena 3D. Oppure integrare dei meccanismi di riconoscimento di un piano per poter diminuire l'errore delle stime calcolate;
- implementare un sistema capace definire in autonomia la locazione spaziale dei corner appartenenti all'arena, attualmente definiti arbitrariamente;
- migliorare il sistema di rendering in modo da diminuire il numero di draw call velocizzando i tempi di render;
- permettere all'utente di modificare dei parametri, come l'area dell'arena simulata o la dimensione dei marker reali, prima di inizializzare la sessione in realtà aumentata.

Bibliografia

[BBBG19] Andrew Banks, Ed Briggs, Ken Borgendale, and Rahul Gupta. MQTT Version 5.0. Oasis standard, OASIS, March 2019. Published 07 March 2019.

[BPV15] Jacob Beal, Danilo Pianini, and Mirko Viroli. Aggregate programming for the internet of things. *Computer*, 48(9):22–30, 2015.

[Cab26] Ricardo Cabello. Three.js: Javascript 3d library. `https://threejs.org`, 2010-2026.

[CVAP22] Roberto Casadei, Mirko Viroli, Gianluca Aguzzi, and Danilo Pianini. Scafi: A scala dsl and toolkit for aggregate programming. *SoftwareX*, 20:101248, 2022.

[Fal] Damiano Falcioni. js-aruco2. `https://github.com/damianofalcioni/js-aruco2`. JavaScript library for ArUco marker detection.

[GJMSMCMC15] Sergio Garrido-Jurado, Rafael Muñoz-Salinas, Francisco Madrid-Cuevas, and Rafael Medina-Carnicer. Generation of fiducial marker dictionaries using mixed integer linear programming. *Pattern Recognition*, 51, 10 2015.

[Imm24] Immersive Web Community Group. WebXR Raw Camera Access Module. `https://immersive-web.github.io/raw-camera-access/`, 2024. Editor's Draft, W3C Community Group.

[pro25] Project Emerge. `https://github.com/Project-Emerge/Project-Emerge-system`, 2025.

[rob25] Robot Emulation. `https://github.com/Project-Emerge/robot-emulation`, 2025. Emulatore virtuale di uno sciame di droni.

[RRMSMC18] Francisco Romero-Ramirez, Rafael Muñoz-Salinas, and Rafael Medina-Carnicer. Speeded up detection of squared fiducial markers. *Image and Vision Computing*, 76, 06 2018.

[Wor19] World Wide Web Consortium (W3C). Webxr device api. `https://www.w3.org/TR/webxr/`, 2019. W3C Candidate Recommendation.

Ringraziamenti

Vorrei ringraziare il Prof. Mirko Viroli e il Dott. Gianluca Aguzzi per avermi seguito con attenzione e disponibilità durante tutto il percorso. I loro consigli e il confronto continuo hanno reso questo lavoro più chiaro, aiutandomi a superare i momenti di incertezza e trasformando le difficoltà in occasioni di crescita.
Un sentito grazie alla mia famiglia, per la fiducia, il supporto e la comprensione dimostrata giorno dopo giorno. Sapere di poter contare su di loro è stato prezioso, soprattutto nei momenti più intensi e impegnativi.
Ringrazio i miei colleghi di corso, compagni di studio e di lunghe sessioni in aula: condividere appunti, dubbi e momenti di leggerezza ha reso questo percorso più leggero e piacevole.
Infine, un grazie ai miei amici di sempre, che anche a distanza hanno saputo sostenermi e ricordarmi di mantenere l'equilibrio, nei momenti in cui il traguardo sembrava ancora lontano.