ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

Corso di Laurea in Ingegneria e Scienze Informatiche

Sviluppo di un sistema di visione artificiale per la rilevazione e la stima della posa basata su marker ArUco in Java

Tesi di laurea in:
Programmazione ad oggetti

Relatore
Prof. Mirko Viroli

Candidato
Bacchini Lorenzo

Correlatore
Dott. Gianluca Aguzzi

III Sessione di Laurea
Anno Accademico 2023-2024

Sommario

Max 2000 characters, strict.

Optional. Max a few lines.

Indice

Elenco delle figure

List of Listings

Capitolo 1

Introduzione

questo capitolo lo scriverò alla fine ma devo ricordarmi di dire che il mio contributo al progetto è stato particolare perché in un ambiente outdoor non sarebbe stato necessario l'utilizzo di marker aruco e camere, ma sarebbe bastato il GPS

Write your intro here.

Bacchini Lorenzo: Add sidenotes in this way. They are named after the author of the thesis

You can use acronyms that your defined previously, such as If you use acronyms twice, they will be written in full only once (indeed, you can mention the now without it being fully explained). In some cases, you may need a plural form of the acronym. For instance, that you are discussing, you may need both and .

Structure of the Thesis

Bacchini Lorenzo: At the end, describe the structure of the paper

Capitolo 2

Background

2.1 Visione artificiale

2.1.1 Cos'è la visione artificiale?

Quando parliamo di visione artificiale o computer vision stiamo considerando un insieme di processi e tecniche che hanno come scopo finale quello di trasformare degli input (solitamente foto o video[1]) in una serie di informazioni utili al calcolatore, che possono poi essere utilizzate per prendere decisioni in maniera autonoma, analizzare una situazione o addirittura creare una rappresentazione del mondo reale 3D che ci circonda. [3] [8]

Quanto sopra descritto non è troppo diverso da ciò che i nostri occhi fanno tutti i giorni, ed infatti, la visione artificiale nasce proprio per permettere al calcolatore di "vedere" esattamente come un essere umano, in modo da poter interagire con l'ambiente circostante.

2.1.2 Come funziona la visione artificiale?

Il processo di visione artificiale può essere suddiviso in tre fasi principali:

1. Acquisizione di un'immagine

2. Interpretazione e analisi dell'immagine

[1] gli input potrebbero essere anche generati da scanner, sensori LiDaR, radar ecc.

3. Richiesta di informazioni sull'immagine analizzata

Nella fase di rilevazione come sopra citato è possibile utilizzare diversi tipi di strumenti come fotocamere o videocamere, ma è nella fase centrale che il processo può differire maggiormente, infatti, l'interpretazione dell'immagine viene effettuata secondo algoritmi che possono essere anche molto diversi in base al loro scopo, negli ultimi anni inoltre si stanno facendo largo nuove tecnologie[2] come l'intelligenza artificiale, il machine learning e il deep learning per poter intraprendere decisioni e svolgere compiti in modo autonomo senza il bisogno dell'intervento umano.

2.1.3 Applicazioni e finalità

Di seguito sono elencate alcune applicazioni della visione artificiale:

- Classificazione di immagini
- Identificazione di oggetti
- Suddivisione di immagini in sezioni da analizzare
- Riconoscimento facciale
- Rilevazione e riconoscimento dei sentimenti di un soggetto
- Ricostruzione di ambienti 3D
- Guida autonoma

La lista dei possibili utilizzi è ovviamente molto vasta ma quelli riportati sopra sono tra i più gettonati sia in ambito professionale che di ricerca.

2.1.4 Principali criticità

Tutte le operazioni che caratterizzano un sistema di visione artificiale possono essere largamente influenzate da una serie di condizioni interne o esterne con il risultato che il nostro sistema potrebbe non operare come previsto.

[2]con il termine nuove non si intende che tecnologie come l'intelligenza artificiale o il machine learning siano state sviluppate negli ultimi anni, ma che iniziano ad essere prese sempre più in considerazione nell'ambito della visione artificiale

Un esempio di condizioni esterne che possono influenzare il comportamento del nostro sistema sono sicuramente l'illuminazione, la prospettiva ed eventuali occlusioni dell'immagine in input, che possono portare ad una maggiore difficoltà di rilevazione e riconoscimento, per quanto riguarda invece i parametri interni possiamo considerare la risoluzione della camera che stiamo utilizzando, l'algoritmo di elaborazione e la complessità (in termini di numero di pixel da elaborare) dell'immagine ottenuta, come parametri che possono variare anche di molto la velocità e la precisione del nostro sistema.

2.1.5 Cenni storici

I primi articoli prodotti riguardanti la visione artificiale risalgono agli anni '60 dove però l'idea di poter acquisire immagini ed elaborarle, facendone comprendere il contenuto all'elaboratore era ancora troppo precoce per l'hardware a disposizione, solo intorno agli anni '80 si sono iniziati a vedere i primi sviluppi significativi grazie all'introduzione della trasformata di Hough e dei primi algoritmi di riconoscimento ottico dei caratteri optical character recognition (OCR). Dagli anni '90 sino ai primi anni '00 l'attenzione si è spostata sullo sviluppo di algoritmi di machine learning, questo ha permesso nel 2001 di sviluppare il primo algoritmo di riconoscimento facciale.[11]

Ad oggi la visione artificiale adotta tecniche e processi completamente differenti rispetto a quelli visti nei suoi primi anni di sviluppo, facendo largo uso di reti convoluzionali e dell'intelligenza artificiale (ormai largamente utilizzabile grazie alla sempre crescente potenza di calcolo dei dispositivi) che le permettono non solo di essere più veloce ma anche di garantire una precisione dei risultati molto maggiore grazie anche al vasto numero di dati a disposizione.

2.2 Marker fiduciari

I marker fiduciari sono degli oggetti che, posti all'interno del campo visivo di una fotocamera possono essere utilizzati come punti di riferimento.

Gli scopi principali di questi marker sono sicuramente la calibrazione della camera, la localizzazione, il tracking e la rilevazione di oggetti.

Alcuni possibili tipi di marker fiduciari sono riportati nella figura 2.1:

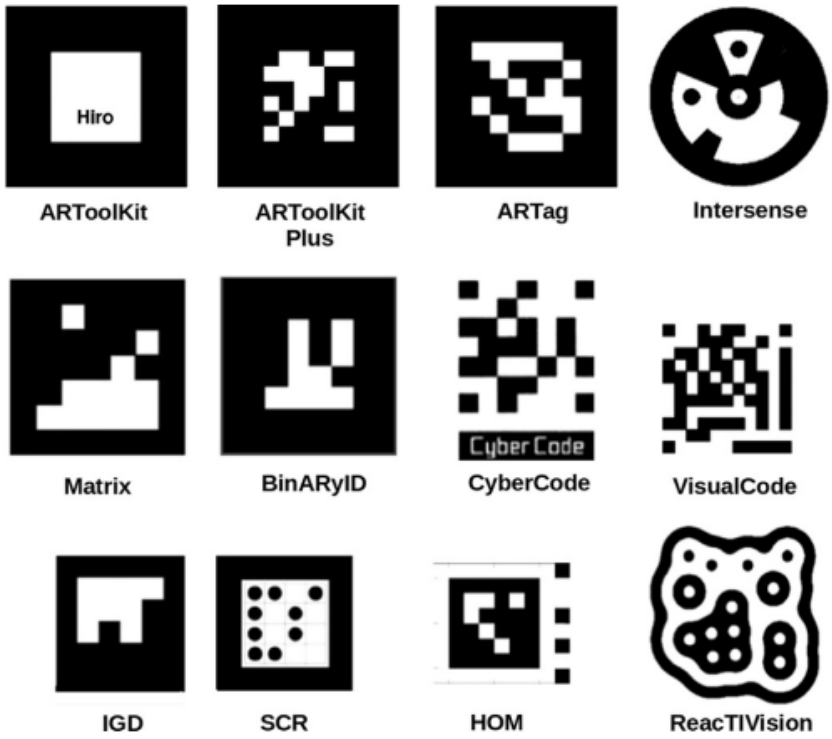

Figura 2.1: Marker fiduciari[6]

2.2.1 Applicazioni

- **Fisica**: per ottenere posizioni e riferimenti di oggetti
- **Realtà aumentata**: utilizzo dei marker come "ancore" così da sapere dove posizionare gli elementi virtuali nel mondo reale
- **Circuiti stampati**: identificano dei pattern così che i macchinari possano operare sui circuiti in maniera autonoma selezionando i componenti univocamente

  Vedi video che riporta l'utilizzo di marker fiduciari per un'applicazione di realtà aumentata: link al video

2.2.2 ArUco markers

I marker Augmented Reality University of Cordova (ArUco) sono una tipologia di marker fiduciari binari molto utilizzata in ambiente di visione artificiale.

La caratteristica principale di questo tipo di marker risiede nel fatto di essere estremamente versatile, dove, con il termine versatile si vuole sottolineare il fatto che questa tipologia di marker non necessita di particolari condizioni di luce ne di grandi capacità di elaborazione per essere processata, in quanto, la complessità di ogni marker può variare in base alle esigenze, vedi figura 2.2.

Un altro aspetto fondamentale degli ArUco marker è dato dal fatto che, attraverso sistemi di posa della camera, utilizzata per ottenere le posizioni dei marker, è possibile calcolare anche la loro rotazione.

Come funzionano?

All'interno del marker abbiamo dei quadrati bianchi e neri che, in base alla loro disposizione identificano univocamente il marker stesso, il numero di questi quadrati e la loro disposizione possono variare in base al "dizionario" selezionato

Dizionario

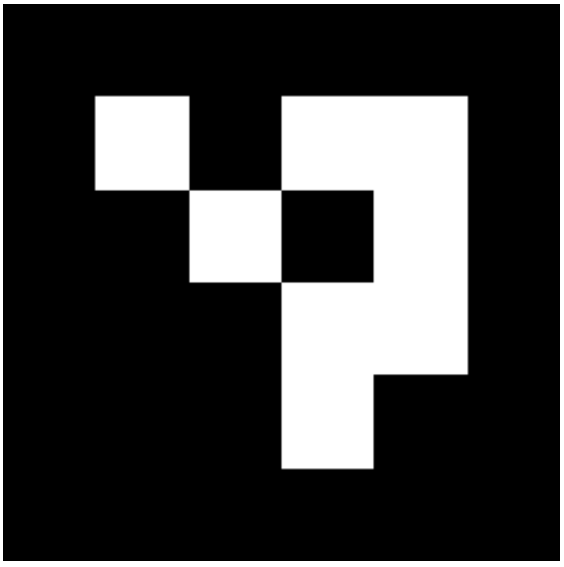

Figura 2.2: ArUco Marker 4x4 id: 0

Ogni ArUco marker appartiene ad un dizionario il quale ne specifica la dimensione e la disposizione interna dei quadrati bianchi e neri, modificando il dizionario possiamo ottenere marker più grandi e complessi che consentono rilevazioni più precise, riducendo la possibilità di confondere tra loro due marker simili, d'altro canto utilizzando marker più piccoli e semplici, nonostante si vada incontro ad una riduzione della precisione, si guadagna in velocità di elaborazione. [9]

DICT_4X4_100

2.3 OpenCV

Open Source Computer Vision Library (OpenCV) è una libreria realizzata inizialmente da Intel pensata per fornire funzioni di visione artificiale che facilitassero il compito degli sviluppatori, garantendo loro un'infrastruttura stabile sulla quale poter lavorare 2.3.

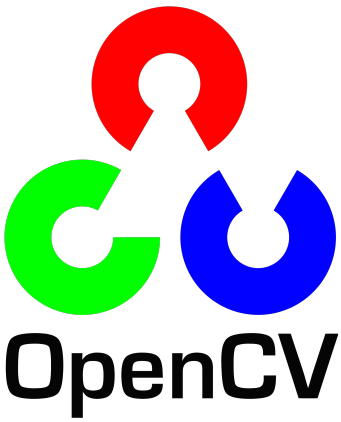

Figura 2.3: logo OpenCV

OpenCV è una libreria multi-piattaforma rilasciata su una licenza open-source, questo ha portato durante gli anni ad una crescita continua grazie anche all'aiuto della community, che ha permesso alla libreria in questione di diventare una tra le più importanti nel suo ambito. [2] [10]

2.3.1 JavaCV

JavaCV è un wrapper per la libreria OpenCV, in grado di fornire, oltre a tutte le funzionalità di OpenCV, anche molti metodi per processare immagini e video, questo fa si che con una singola dipendenza si sia in grado di utilizzare tutte le funzionalità in maniera agevolata attraverso il wrapper, senza dover interagire con il codice nativo di OpenCV [1]

2.4 Aggregate Computing

Prima di poterci addentrare nel concetto di aggregate computing è importante spendere due parole per introdurre i sistemi distribuiti dai quali poi saremo in grado di derivare l'idea alla base dell'aggregate computing.

2.4.1 Sistema distribuito

Un sistema distribuito rappresenta una rete di processi o "nodi di elaborazione" che svolgono calcoli individualmente ma che appaiono dall'esterno come un unico sistema.

Questo tipo di approccio è caratterizzato da una grande eterogeneità sia a livello hardware che software, la quale porta il sistema ad essere aperto a tecnologie diverse, che comunicano tra loro attraverso lo scambio di messaggi per far apparire il sistema distribuito come un'entità unica.

L'aspetto fondamentale di questi sistemi riguarda il fatto che ad ogni singolo nodo viene affidato un compito da espletare, il nodo in questione può comunicare con gli altri al fine di portare a termine il suo incarico e può essere a sua volta interrogato dagli altri per fornire i risultati della sua elaborazione. Tutto questo può essere visto dall'esterno come un unico sistema che però al suo interno è composto da una serie di dispositivi con compiti differenti che contribuiscono al raggiungimento di uno scopo comune 2.4

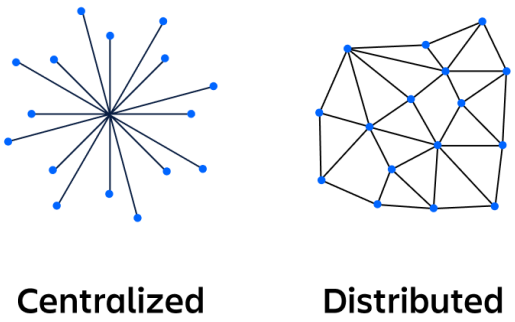

Figura 2.4: Sistema centralizzato Vs distribuito [13]

2.4.2 Aggregate computing

Il termine Aggregate computing rappresenta un approccio emergente nell'ambito della gestione di sistemi distribuiti complessi [12] che sposta l'attenzione dal calcolo effettuato da un singolo nodo al far emergere un comportamento globale del sistema sulla base dei calcoli effettuati sui singoli nodi.

A primo avviso la differenza tra i due sembra marginale ma se scendiamo più nel dettaglio possiamo notare che un sistema distribuito focalizza la propria attenzione sulla suddivisione dei compiti ai nodi, che possono comunicare tra loro per ottenere un fine comune mentre un sistema di aggregate computing pone maggiore enfasi sullo scambio di informazioni tra nodi vicini, che, comunicando tra loro, fanno emergere una serie di comportamenti complessi che descrivono il sistema nella sua totalità. Nel contesto di aggregate computing un nodo non è in grado di comunicare con tutti i nodi, ma solamente con quelli a lui vicini, attraverso i quali può progredire nel suo compito e derivare informazioni sulla rete globale, questo suggerisce quindi che il comportamento generale del sistema visto dall'esterno e guidato da interazioni prettamente locali al suo interno.

La cosa importante da ricordare quando si parla di aggregate computing e che un gruppo di nodi locali che comunicano, con il fine di ottenere una serie di risultati, potrebbe non conoscere l'intero sistema che lo circonda

Vantaggi

I vantaggi offerti dall'approccio dell'aggregate computing sono molteplici e spesso dipendono proprio dalla modalità operativa del sistema, in questo caso quindi dalla possibilità di derivare comportamenti emergenti sulla base di interazioni locali:

- Scalabilità
- Tolleranza ai guasti
- Auto-organizzazione

Questi vantaggi sono dovuti al fatto che ogni nodo comunica solo con i suoi vicini, di conseguenza non conosce l'intera rete, non è quindi importante che la rete resti invariata perché l'eventuale aggiunta o rimozione di un componente non altera il sistema visto dall'esterno grazie alla capacità dei nodi di riorganizzarsi

Applicazioni

Le principali applicazioni di sistemi di aggregate computing possono essere ricercate oggi in ambiti quali:

- Reti di sensori
- Robotica collettiva
- Smart cities
- sistemi IoT

dove con reti di sensori indichiamo una serie di sensori che vengono utilizzati per monitorare una certa area geografica della quale vogliamo estrarre informazioni in maniera aggregata quali temperatura, pressione atmosferica ecc.

Per robotica collettiva si intende tutta quella serie di applicazioni che consentono a robot e droni di auto-organizzarsi e coordinarsi secondo sciami o schemi predefiniti.

Con smart cities e sistemi IoT invece possiamo riassumere le operazioni volte a gestire l'infrastruttura cittadina come l'illuminazione, il controllo dei semafori e del traffico, ma anche sistemi IoT decentralizzati che interagiscono tra loro per organizzarsi e prendere decisioni in autonomia.

2.5 Calibrazione della camera

Di seguito sono riportate alcune informazioni di base per comprendere il concetto di calibrazione della camera

2.5.1 Cos'è la calibrazione?

La calibrazione è un processo volto a ottenere i parametri della lente e del sensore della videocamera/fotocamera, tali parametri potranno poi essere impiegati per correggere la distorsione della lente, calcolare distanze, ricostruire l'immagine 3D di partenza ecc [7].

2.5.2 Modello di calibrazione

Esistono diversi modelli di calibrazione in base al tipo di camera che si intende utilizzare, nel nostro caso possiamo ipotizzare l'utilizzo di uno dei modelli più semplici e cioè quello di una fotocamera con obiettivo stenopeico 2.5, nella quale però teniamo conto anche della distorsione della lente. [5] [4]

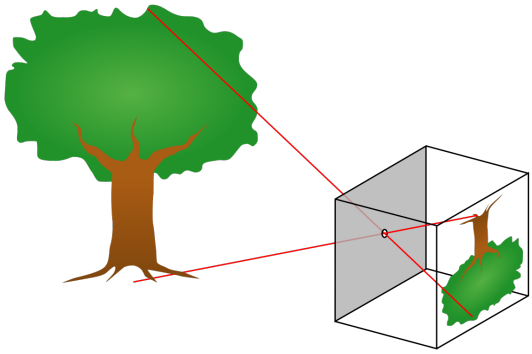

Figura 2.5: Rappresentazione camera con obiettivo stenopeico

Il parametro più importante è la matrice della camera "camera matrix" che mappa le coordinate del mondo 3D nell'immagine 2D catturata, questa matrice è ottenuta a partire dai parametri intrinseci e estrinseci:

- Parametri estrinseci: rappresentano la posizione e rotazione della camera nel mondo reale.

- Parametri intrinseci: consentono di trasformare le coordinate della camera 3D nelle coordinate dell'immagine 2D.

l'obiettivo della calibrazione è proprio quello di determinare questi due parametri così da poter calcolare la matrice della camera. 2.6

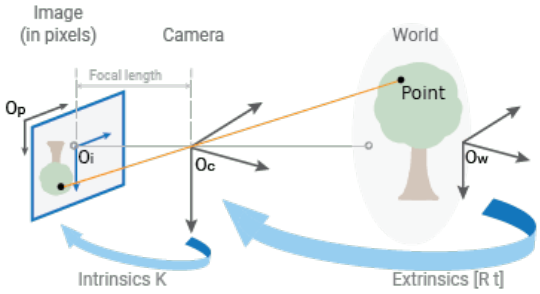

Figura 2.6: Processo di conversione dal mondo reale 3D all'immagine 2D

2.5.3 Distorsione della lente

Un altro aspetto fondamentale da tenere in considerazione riguarda la distorsione introdotta dalla lente della camera, nel modello utilizzato infatti si trascura l'uso della lente ma in situazioni reali l'obiettivo gioca un ruolo cruciale, poiché introduce due tipi di distorsione:

- Distorsione radiale: condizione per la quale i raggi di luce vengono curvati in maniera non uniforme, deformando l'immagine finale a partire dal centro dell'obiettivo. 2.7

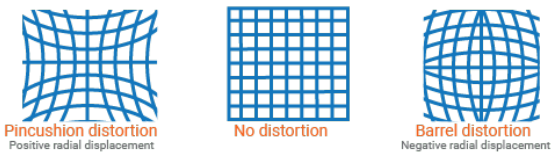

Figura 2.7: Distorsione radiale

- Distorsione tangenziale: distorsione che ha luogo nel caso in cui la lente e il piano che si vuole fotografare non sono paralleli. 2.8

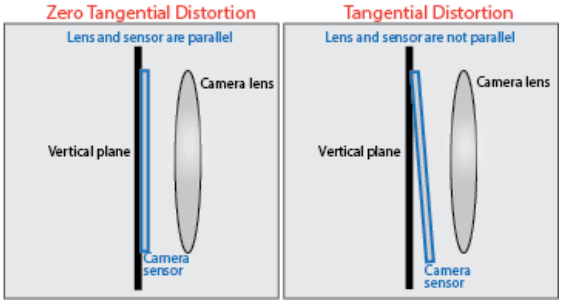

Figura 2.8: Distorsione tangenziale

Capitolo 3

Analisi

Il progetto si pone l'obiettivo di creare un sistema software sviluppato in java o altro linguaggio che sia in grado di essere eseguito sulla Java Virtual Machine (JVM), con il compito di localizzare e tracciare gli spostamenti di alcuni dispositivi[1] nel campo visivo di una videocamera.

Per poter analizzare correttamente il sistema in questione è necessario però fare un passo indietro, per introdurre il contesto nel quale dovrà inserirsi:

> l'argomento di questa tesi rappresenta solo un componente di un progetto più complesso, sviluppato dai ricercatori dell'università di Bologna, con l'obiettivo di gestire una serie di dispositivi (robot, droni ecc.), connessi tra loro, secondo il paradigma dell'aggregate computing. Questo approccio permette ai dispositivi della rete di auto-organizzarsi presentando le caratteristiche tipiche della robotica degli sciami.
>
> Il vincolo riguardante il linguaggio da utilizzare, viene dal fatto che il software attualmente usato dai ricercatori è eseguito dalla JVM e di conseguenza per avere un'interazione più semplice tra i due componenti è importante che anche la parte di visione artificiale sia scritta con un linguaggio compatibile.

L'aspetto che ha portato alla stesura di questa tesi, si deve ricercare nel fatto che questo tipo di sistema deve poter funzionare anche in applicazioni indoor, nelle quali, l'utilizzo del GPS per tenere traccia degli spostamenti dei dispositivi

[1] oggetti di ogni genere e tipo dei quali si vuole conoscere posizione e rotazione chiamati genericamente dispositivi in questa dissertazione

è ostacolato da numerosi fattori quali pareti e strutture che possono ridurne il segnale.

3.1 Requisiti funzionali

- Il sistema deve poter rilevare i dispositivi
- Il sistema deve poter calcolare distanza, posizione e angolo di rotazione dei dispositivi rispetto ad un punto fissato nello spazio.
- Il sistema dovrà poter essere interrogato per ottenere i risultati computati riguardo il posizionamento dei dispositivi.

3.2 Requisiti non funzionali

- Il sistema deve poter rilevare i dispositivi ad una distanza di due/tre metri.
- Il sistema deve essere sufficientemente reattivo da rilevare e tracciare movimenti repentini da parte dei dispositivi.
- Il sistema deve rimanere stabile ed affidabile anche in presenza di condizioni di scarsa illuminazione.
- Il sistema deve essere efficiente al punto che il tempo impiegato per stimare la posizione dei dispositivi non influenzi il tempo di esecuzione del software in cui è utilizzato.
- Il sistema deve essere sufficientemente preciso da garantire un corretto funzionamento generale, senza incorrere in rilevazioni fuorvianti che porterebbero al crash dello stesso.

3.3 Requisiti tecnologici

- Il sistema deve essere scritto in java o un linguaggio che sia in grado di essere eseguito dalla JVM.

- La videocamera utilizzata per la rilevazione dei dispositivi deve avere una risoluzione minima FullHD.

3.4 Analisi del dominio

Come già sopracitato, l'oggetto di questa tesi andrà a costituire parte di un sistema più ampio, il quale, una volta assemblato, sarà in grado di:

1. Rilevare e tracciare gli spostamenti dei dispositivi.
2. Inviare ai vari nodi le informazioni acquisite durante la fase di rilevazione.
3. Far comunicare tra loro i nodi vicini.
4. Far organizzare in maniera autonoma i diversi dispositivi attraverso le informazioni raccolte, secondo algoritmi di aggregate computing senza quindi il bisogno di elaborare gli spostamenti e le nuove posizioni in maniera centralizzata.

Vedi diagramma esemplificativo 3.1

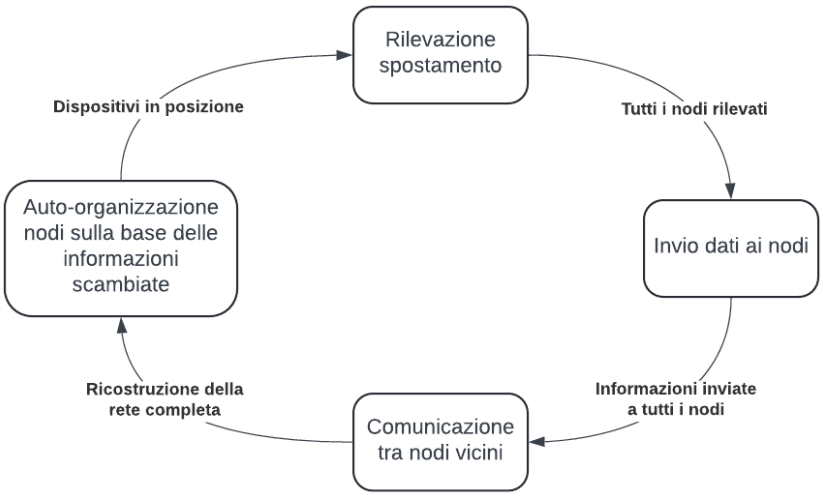

Figura 3.1: Diagramma a stati semplificato del dominio

Capitolo 4

Design

L'ambito di questa tesi è focalizzato sullo sviluppo della componente di visione artificiale utile a rilevare e seguire i dispositivi nel campo visivo della videocamera.

Nelle seguenti sezioni vengono illustrate le scelte, operate in osservanza dei requisiti enunciati nelle sezioni 3.1 e 3.2 che costituiranno poi l'implementazione del software oggetto della tesi.

Prima però di poter approfondire l'architettura del sistema è necessario stabilire come rilevare e identificare quelli che finora abbiamo genericamente chiamato "dispositivi".

4.1 ArUco markers

Dopo un'attenta ricerca volta a determinare quale fosse il metodo più efficacie per poter identificare un qualsiasi oggetto nel campo visivo di una videocamera mi sono imbattuto nei marker ArUco 2.2.2, una tipologia di marker fiduciari 2.2 particolarmente utilizzati in contesti di visione artificiale.

La motivazione che mi ha spinto ad utilizzare i marker ArUco deriva dai numerosi vantaggi che questa tecnologia offre, per prima cosa questo tipo di marker è estremamente semplice e scalare, é infatti possibile scegliere fra numerose versioni che si differenziano in base al dizionario utilizzato 2.2.2, inoltre i marker ArUco garantiscono una buona affidabilità anche in condizioni di scarsa illuminazione, scenario da non sottovalutare in una applicazione di visione artificiale, infine un

altro vantaggio è sicuramente dato dalla loro velocità di rilevazione che permette quindi l'uso di un gran numero di marker sulla scena senza rallentamenti.

Per poter utilizzare i marker ArUco è stata utilizzata la libreria offerta da OpenCV 2.3

4.2 Videocamera

Per poter rispettare i requisiti imposti in fase di analisi riguardanti la videocamera che il sistema avrebbe dovuto utilizzare per ottenere il flusso video, la scelta è ricaduta su una webcam FullHD capace di registrare a 30/60fps, per contenere i costi, garantire una risoluzione sufficiente al riconoscimento degli oggetti e avere un campo visivo ampio.

4.3 Architettura

Premessa: *L'architettura riguarderà il solo sistema di visione artificiale, realizzato da me come software stand-alone che è poi stato rielaborato dai ricercatori per poterlo integrare con il loro progetto, non verrà trattata l'architettura del sistema finale se non attraverso alcuni cenni utili a comprendere le scelte intraprese.*

Tutti i diagrammi che seguono descrivono l'architettura del sistema, esclusi quelli delle classi, riportano solo i metodi principali e non sono quindi esaustivi.

Finora sono state presentate soluzioni che riguardano la videocamera e l'utilizzo di marker ArUco per rilevare e identificare gli oggetti nella scena, dal punto di vista del software invece per ottenere risultati conformi ai requisiti e di conseguenza un sistema capace di operare anche in condizioni di scarsa luminosità, con un buon livello di precisione e a distanze considerevoli, si è reso necessario l'impiego di sistemi di calibrazione della videocamera oltre che ovviamente a funzionalità per determinare la posizione degli oggetti.

Il software da me realizzato si compone di tre parti principali descritte da tre classi java:

- App.java

- CameraCalibrator.java
- CameraPose.java

`App.java` si occupa di orchestrare le diverse fasi di avvio dell'applicazione, consentendo in una fase iniziale di calibrare la camera attualmente in uso attraverso la classe `CameraCalibrator.java` e in un secondo momento di calcolare la posa della camera rispetto ai marker sulla scena con la classe `CameraPose.java` 4.1.

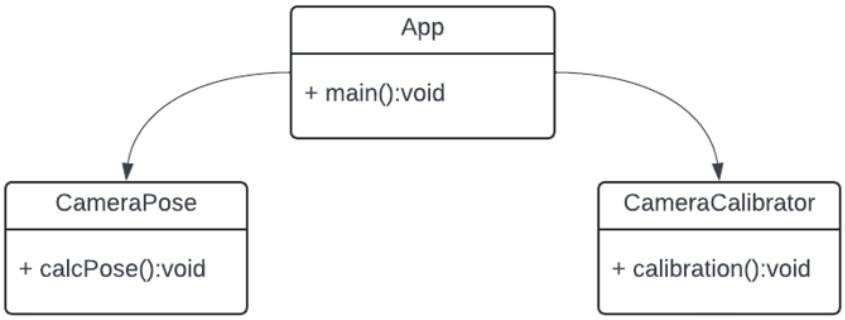

Figura 4.1: Architettura del sistema

4.4 Design dettagliato

Entrando più nel dettaglio dell'architettura, il sistema si compone anche di altre due classi denominate: `ResolutionEnum.java` e `InputParameters.java` andando quindi a definire il sistema nella sua totalità come in figura: 4.2 dove le due classi sono rispettivamente una enum che racchiude tutte[1] le risoluzioni che la camera può utilizzare ed una Graphical User Interface (GUI) che permette all'utente di configurare il software in base alle proprie preferenze in fase di avvio.

4.4.1 App

Questa classe rappresenta il punto di ingresso dell'applicazione e si occupa di istanziare tutte le classi necessarie per il corretto funzionamento del sistema all'interno del main. 4.3

[1]sono riportate solo le risoluzioni più comuni

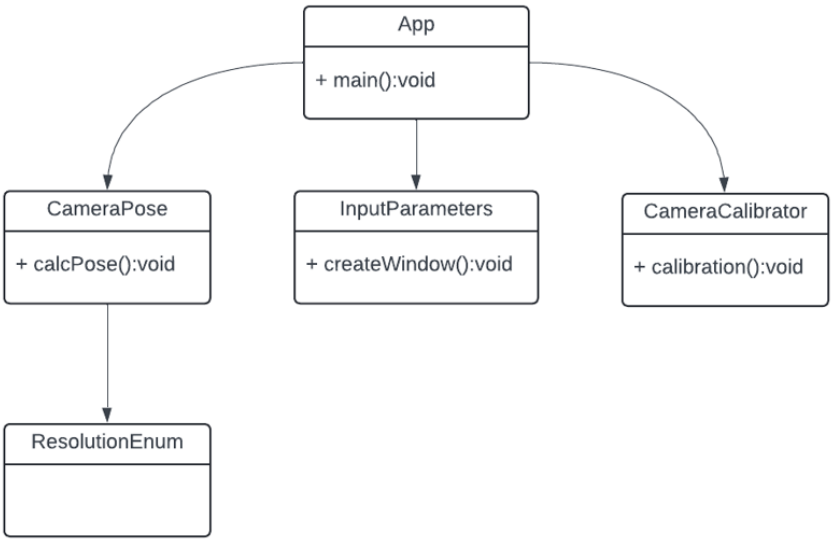

Figura 4.2: Architettura completa

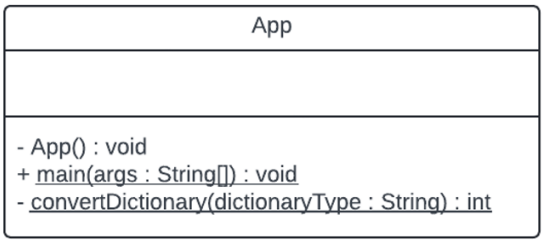

Figura 4.3: Classe App.java

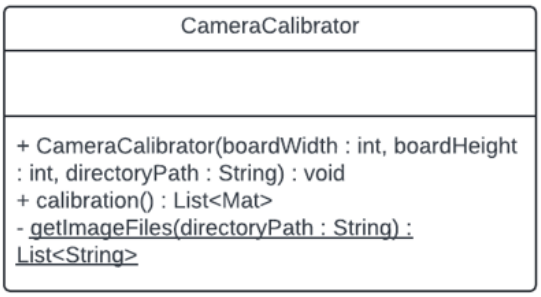

Figura 4.4: Classe CameraCalibrator.java

4.4.2 cameraCalibrator

La classe `CameraCalibrator.java` ricopre un ruolo molto importante, il suo obiettivo è quello di calibrare la camera 2.5 affinché le immagini ottenute consentano di stabilire correttamente la posizione della stessa rispetto agli oggetti nella scena. 4.4.

4.4.3 cameraPose

La classe `CameraPose.java` rappresenta il fulcro del progetto, grazie ai metodi sviluppati è possibile ottenere la posa della camera rispetto agli oggetti identificati dai marker ArUco. Le informazioni ricavabili riguardano distanza, posizione e rotazione dei marker, inoltre è anche possibile visualizzarne graficamente lo spostamento e l'orientamento. 4.5

4.4.4 resolutionEnum

Enumerazione in grado di fornire i formati più utilizzati per flussi video. 4.6

4.4.5 inputParameters

La classe `InputParameters.java` rappresenta una GUI con la quale l'utente finale può interagire, al fine di configurare il programma in base alle proprie specifiche e preferenze. 4.7

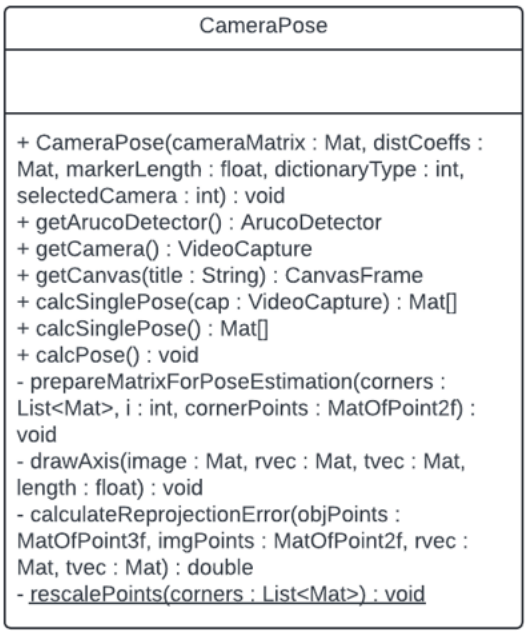

Figura 4.5: Classe CameraPose.java

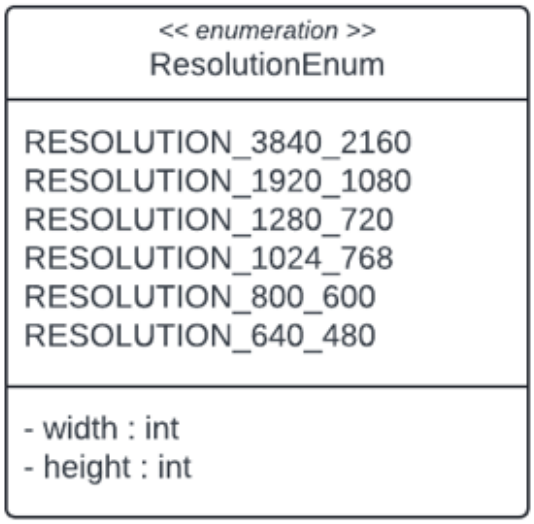

Figura 4.6: Classe ResolutionEnum.java

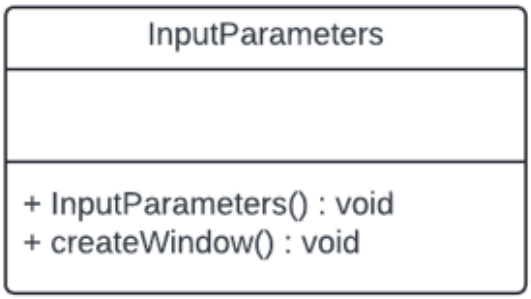

Figura 4.7: Classe InputParameters.java

Figura 4.8: Associazione ResolutionEnum.java

4.4.6 cameraPoseBlock

Nell'immagine 4.8 viene mostrata l'associazione tra la classe `CameraPose.java` e la classe `ResolutionEnum.java`, grazie alla quale la prima riesce ad ottenere l'elenco delle possibili risoluzioni video utilizzate dalla camera.

4.4.7 UI

Nell'associazione in figura 4.9 `App.java` fa uso della classe `InputParameters.java` per poter istanziare correttamente i parametri utili al software per adattarsi all'ambiente in cui il sistema viene impiegato.

Figura 4.9: Associazione InputParameters.java

Capitolo 5

Implementazione

In questo capitolo verranno presentate alcune scelte implementative effettuate durante lo sviluppo software, con particolare attenzione alle fasi di calibrazione e posa della camera.

5.1 Creazione del progetto

Il software è stato sviluppato all'interno di un progetto Java utilizzando Gradle con uno script di build in Kotlin.

5.2 OpenCV

Per poter utilizzare funzionalità di visione artificiale, mi sono affidato alla libreria OpenCV 2.3.

OpenCV mette a disposizione una serie di moduli, attraverso i quali, si possono realizzare applicazioni di visione artificiale di ogni tipo: rilevazione di oggetti, riconoscimento facciale, ecc.

5.2.1 JavaCV

JavaCV 2.3.1 è un wrapper di OpenCV, durante l'implementazione è infatti stato usato al posto di OpenCV nativo in quanto fornisce classi e metodi semplificati

```
implementation ("org.bytedeco:javacv-platform:1.5.10")
```

```
static {
  Loader.load(opencv_java.class);
}
```

per la creazione della GUI come `CanvasFrame` e classi di utility sviluppate per l'utilizzo in java.

La dipendenza necessaria per utilizzare JavaCV è stata inserita nel file build.gradle.kts sezione 5.2.1

5.3 Avvio dell'app

La classe `App.java` è responsabile dell'inizializzazione dei parametri del sistema oltre che dell'avvio della fase di calibrazione e posa della camera.

Come prima operazione vengono caricate le classi di OpenCV in maniera dinamica sezione 5.3, dopodiché `App.java` si occuperà di istanziare e lanciare la classe `InputParameter.java` utile a configurare i parametri dell'app come mostrato dal seguente codice: sezione 5.3, infine sarà il turno di calibrazione e posa della camera, che, come è possibile notare da sezione 5.3 vengono chiamate sequenzialmente al fine di passare i parametri della camera 2.5 alla fase di posa.

```
final InputParameters ip = new InputParameters();
ip.createWindow();

// Wait for the user to set the parameters
final int squaresX = ip.getSquaresX();
final int squaresY = ip.getSquaresY();
final float markerLength = ip.getMarkerLength();
final String directoryPath = ip.getDirectoryPath();
final int selectedCamera = ip.getCameraIndex();
final int dictionaryType = convertDictionary(ip.getDictionaryType());
// Close the input parameters window
ip.close();
```

```
//Camera calibration
final CameraCalibrator cc = new CameraCalibrator(squaresX, squaresY, directoryPath
    );
final List<Mat> cameraParam = cc.calibration();

//Camera pose estimation
final CameraPose cp = new CameraPose(cameraParam.get(0), cameraParam.get(1),
  markerLength, dictionaryType, selectedCamera);
cp.calcPose();
```

5.4 Calibrazione

La fase di calibrazione della camera rappresenta un aspetto fondamentale del sistema finale in quanto consente di ottenere risultati precisi e affidabili nelle fasi successive.

Per effettuare la calibrazione OpenCV mette a disposizione varie strategie, è possibile infatti utilizzare una scacchiera classica, oppure una scacchiera o una semplice tabella di marker ArUco, nel sistema sviluppato si è optato per la prima soluzione, una scacchiera classica infatti offre un'alta precisione in situazioni ideali e semplicità di implementazione, fattori determinanti visto l'utilizzo di immagini pre acquisite ottenute in condizioni ottimali.

Il ruolo principale di questa classe quindi, è quello di restituire alla chiamata del metodo `Calibration()` la matrice della camera e i coefficienti di distorsione della lente, per fare ciò sono stati utilizzati i metodi di JavaCV come mostrato nei seguenti frammenti di codice.

5.4.1 Rilevazione scacchiera

Nel codice sezione 5.4.1 si è fatto uso del metodo `findChessBoardCorners()` per ottenere le posizioni degli angoli interni della scacchiera, sulla base di un'immagine della stessa convertita in bianco e nero per ridurre la complessità computazionale.

```
final Mat image = Imgcodecs.imread(filePath);
final Mat grayImage = new Mat();
//Converting to a gray scale image to reduce computational complexity
Imgproc.cvtColor(image, grayImage, Imgproc.COLOR_BGR2GRAY);

final MatOfPoint2f imageCorners = new MatOfPoint2f();
//Scanning the image to extract the corners
final boolean found = Calib3d.findChessboardCorners(grayImage, boardSize,
    imageCorners);
```

```
//finishing the corners with the cornerSubPix method
if (found) {
  Imgproc.cornerSubPix(grayImage, imageCorners, new Size(WIN_X_SIZE, WIN_Y_SIZE),
        new Size(ZERO_X_ZONE, ZERO_Y_ZONE),
        new TermCriteria(TermCriteria.EPS + TermCriteria.COUNT, MAX_ITERATION,
            ACCURACY));

        imagePoints.add(imageCorners);
        objectPoints.add(objectPoint);
}
```

5.4.2 Rifinitura angoli

Si è ritenuto necessario rifinire le posizioni degli angoli della scacchiera per aumentarne ulteriormente la precisione, analizzando l'immagine a livello di sub-pixel[1] con il metodo `cornerSubPix()` sezione 5.4.2.

5.4.3 Calibrazione

La calibrazione vera e propria è svolta dal metodo `calibrateCamera()`, che oltre a popolare la matrice della camera e dei coefficienti di distorsione, restituisce l'errore quadratico medio di riproiezione dei punti del mondo reale nell'immagine analizzata sezione 5.4.3.

[1] tecnica che permette di analizzare lo spazio vuoto tra i pixel, vedi: `https://www.pomeas.com/download/2021/5/4/what-subpixel/` per maggiori informazioni

```
//calibration of the camera
final double rms = Calib3d.calibrateCamera(objectPoints, imagePoints, boardSize,
    cameraMatrix, distCoeffs, rvecs, tvecs);
```

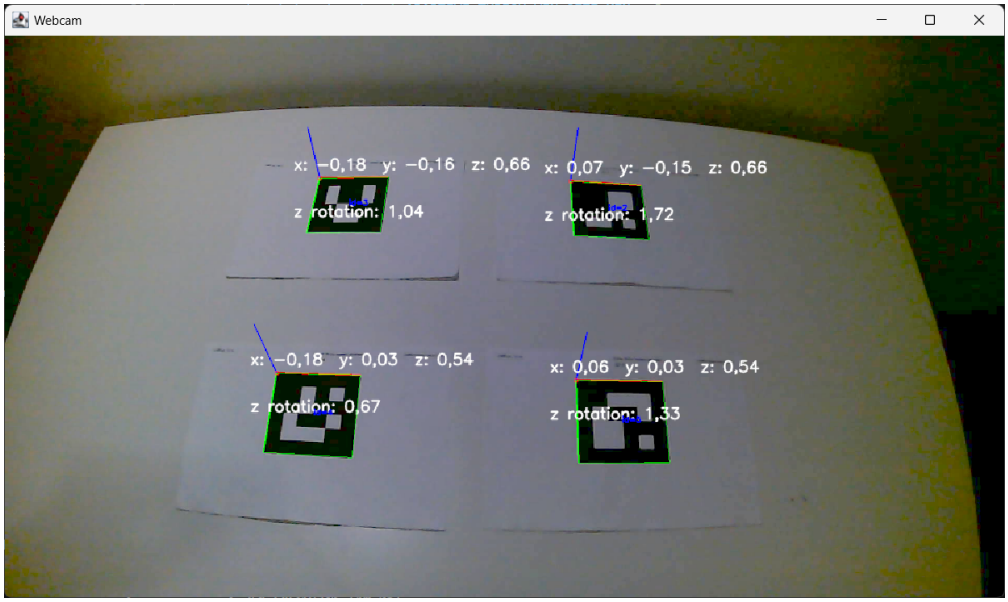

Figura 5.1: Posa della camera

5.5 Posa

La classe `CameraPose.java` si occupa di calcolare la posa della camera rispetto ai marker ArUco e, successivamente, di visualizzare in un ambiente di realtà aumentata la posizione e la rotazione dei marker 5.1.

L'elenco di operazioni che tale classe esegue può essere riassunto in questo modo:

1. Cattura del frame. 5.5.1
2. Rilevazione marker. 5.5.2
3. Inizio posa
   (a) calcolo posa. 5.5.3
   (b) trascrizione posizione e rotazione di ogni marker sul frame
   (c) calcolo errore di riproiezione
4. restituzione frame aumentato[2]

[2]un frame aumentato è un frame al quale sono stati sovrapposti elementi attraverso operazioni di visione artificiale

```
public enum ResolutionEnum {
  RESOLUTION_3840_2160(3840, 2160),
  RESOLUTION_1920_1080(1920, 1080),
  RESOLUTION_1280_720(1280, 720),
  RESOLUTION_1024_768(1024, 768),
  RESOLUTION_800_600(800, 600),
  RESOLUTION_640_480(640, 480);
}
```

5.5.1 Cattura del frame

Durante la fase di cattura del frame ho dovuto adottare alcune scelte a causa di errori nella stima della posa della camera:

Risoluzione

La classe utilizzata per la cattura del frame apriva la camera con una risoluzione standard, la quale però non combaciava con quella delle immagini utilizzate per la calibrazione, questa discrepanza faceva si che la successiva fase di posa operasse su immagini di risoluzioni differenti da quelle attese producendo errori nella stima.

Per risolvere tale problema ho creato l'enumerazione `ResolutionEnum.java` riportata in maniera semplificata[3] dal seguente codice sezione 5.5.1 in modo da poter sempre impostare la risoluzione massima possibile della webcam (questo perché è logico pensare di avere a disposizione immagini per la calibrazione alla maggior risoluzione possibile).

Esposizione

La reattività del sistema in termini di capacità di tracciamento degli oggetti è risultata essere scarsa nei primi test qualitativi effettuati, per risolvere tale problema si è operato sul tempo di esposizione della camera, che ha permesso di incrementare notevolmente la reattività a discapito di una riduzione del range operativo[4], rendendo il sistema in grado di rilevare i marker solo in condizioni di luce più elevate. sezione 5.5.1

[3] L'enumerazione presentata al lettore contiene solo alcune risoluzioni tra le quali il sistema può scegliere, la classe utilizzata dal software è più complessa e contiene attributi e metodi per fornire l'altezza e la larghezza di ogni risoluzione

[4] intervallo nel quale uno strumento può operare correttamente

```
//Costants for the camera
private static final int CAMERA_EXPOSURE = -6;

//Setting the camera exposure to reduce the motion blur
capture.set(Videoio.CAP_PROP_EXPOSURE, CAMERA_EXPOSURE);
```

```
//Resize the frame to speed up the marker detection
final Mat reducedFrame = new Mat();
Imgproc.resize(frame, reducedFrame, new Size((double) frame.width() / SCALE, (
    double) frame.height() / SCALE));
```

È possibile modificare il valore dell'esposizione della camera per ottenere un sistema più reattivo decrementandolo, o per poter operare in condizioni di scarsa luminosità incrementandolo.

5.5.2 Rilevazione marker

Durante il calcolo della posa mi sono reso conto che gran parte del tempo veniva impiegato nella fase di rilevazione del marker, rallentando l'intero processo, per risolvere tale problema ho deciso di modificare la dimensione del frame ottenuto prima di rilevare i marker, dividendo altezza e larghezza per un fattore arbitrario sezione 5.5.2, in modo da passare al metodo `detectMarkers()` un'immagine più semplice da elaborare e di conseguenza velocizzare il processo sezione 5.5.2.

Adottando questa soluzione bisogna però fare attenzione a due aspetti:

1. Risoluzione troppo bassa: dividendo il frame ottenuto per un fattore arbitrario si rischia di ottenere immagini di bassa risoluzione, questo può portare ad una perdita significativa della precisione in casi in cui i marker sono piccoli o peggio all'impossibilità di rilevarli.

2. Posizioni degli angoli dei marker falsati: avendo in questo caso effettuato la rilevazione su un'immagine ristretta, le posizioni degli angoli dei marker sa-

```
arucoDetector.detectMarkers(gray, corners, ids);
if (!corners.isEmpty()) {
  rescalePoints(corners);
}
```

```
private static void rescalePoints(final List<Mat> corners) {
  for (final Mat corner : corners) {
    for (int i = 0; i < CORNER_NUMBER; i++) {
      final double[] data = corner.get(0, i);
      data[0] *= SCALE;
      data[1] *= SCALE;
      corner.put(0, i, data);
    }
  }
}
```

```
Calib3d.solvePnP(objPoints, cornerPoints, cameraMatrix,
  new MatOfDouble(distCoeffs), rvec, tvec,
  false, Calib3d.SOLVEPNP_ITERATIVE
);
```

ranno relative a tale immagine, per poter tornare ad utilizzare il frame di partenza nelle successive fasi è sufficiente chiamare il metodo `rescalePoints()` voce 2, in grado di tradurre le posizioni degli angoli dei marker ottenute sull'immagine ristretta nelle coordinate dell'immagine originale.

5.5.3 Calcolo Posa

Questa fase rappresenta il fine ultimo del sistema e cioè stabilire una relazione tra la posizione della camera e la posizione dei marker, grazie ad OpenCV questa operazione può essere svolta con il metodo `solvePnP()` che restituisce i vettori di rotazione e traslazione in grado di tradurre le coordinate 3D nelle coordinate 2D della camera sezione 5.5.3.

Come è possibile vedere da 4.5 la classe `CameraPose.java` mette a disposizione altri due metodi per la stima della posa: `CalcSinglePose(VideoCapture cap)` e `CalcSinglePose()` in grado ottenere la posa per un solo frame e ritornare i parametri che descrivono rotazione, traslazione e id di ogni marker rilevato. I due metodi differiscono solamente per un parametro che rappresenta l'interfaccia usata per ottenere il frame, il metodo che ne è sprovvisto dovrà ad ogni chiamata istanziare un `VideoCapture`, rallentando le successive fasi di calcolo della posa in attesa del frame.

Capitolo 6

Valutazione

Capitolo 7

Conclusioni

Capitolo 8

Sviluppi futuri

Capitolo 9

Ringraziamenti

I suggest referencing stuff as follows: fig. 9.1 or Figura 9.1

9.1 Some cool topic

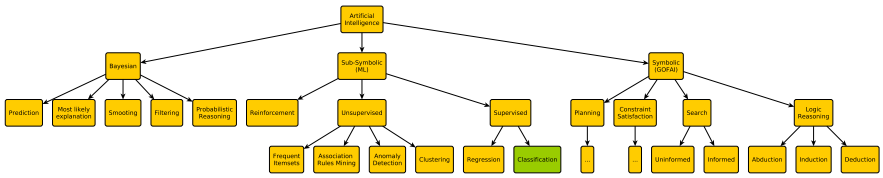

Figura 9.1: Some random image

Capitolo 10

Contribution

You may also put some code snippet (which is NOT float by default), eg: capitolo 10.

10.1 Fancy formulas here

```
public class HelloWorld {
  public static void main(String[] args) {
    // Prints "Hello, World" to the terminal window.
    System.out.println("Hello, World");
  }
}
```

Acronimi

ArUco Augmented Reality University of Cordova. 1

GUI Graphical User Interface. 19

JVM Java Virtual Machine. 13

OCR optical character recognition. 5

Glossario

DICT_4X4_100 dizionario con marker 4x4 e con una quantità massima di marker istanziabili pari a 100. 7

FullHD risoluzione di 1920 x 1080 pixel. 15

nodo dispositivo visto come elemento di una rete. 9

scena campo visivo della videocamera. 18

Bibliografia

[1] Samuel Audet. *javacv*. URL: https://github.com/bytedeco/javacv?tab=readme-ov-file (visitato il 06/10/2024).

[2] D.L. Baggio. *OpenCV 3.0 Computer Vision with Java*. Packt Publishing, 2015. ISBN: 9781783283989. URL: https://books.google.it/books?id=LFtICgAAQBAJ.

[3] G. Bradski e A. Kaehler. *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly Media, 2008, pp. 2–5. ISBN: 9780596554040. URL: https://books.google.it/books?id=seAgiOfu2EIC.

[4] Wikipedia contributors. *Pinhole camera*. 2024. URL: https://en.wikipedia.org/wiki/Pinhole_camera (visitato il 14/10/2024).

[5] Wikipedia contributors. *Pinhole camera model*. 2024. URL: https://en.wikipedia.org/wiki/Pinhole_camera_model (visitato il 13/10/2024).

[6] S. Garrido-Jurado et al. "Automatic generation and detection of highly reliable fiducial markers under occlusion". In: *Pattern Recognition* 47.6 (2014), pp. 2280–2292. ISSN: 0031-3203. DOI: https://doi.org/10.1016/j.patcog.2014.01.005. URL: https://www.sciencedirect.com/science/article/pii/S0031320314000235.

[7] MathWorks. *What Is Camera Calibration?* 2024. URL: https://it.mathworks.com/help/vision/ug/camera-calibration.html (visitato il 13/10/2024).

[8] Microsoft. *Che cos'è la visione artificiale?* URL: https://azure.microsoft.com/it-it/resources/cloud-computing-dictionary/what-is-computer-vision#classificazione-degli-oggetti (visitato il 22/09/2024).

[9] Brandon Minor. *Reverse-Engineering Fiducial Markers For Perception*. URL: https://www.tangramvision.com/blog/reverse-engineering-fiducial-markers-for-perception (visitato il 22/09/2024).

[10] OpenCV.org. *OpenCV*. URL: https://opencv.org (visitato il 06/10/2024).

[11] The Sama Team. *Computer Vision: History & How it Works*. URL: https://www.sama.com/blog/computer-vision-history-how-it-works (visitato il 22/09/2024).

[12] Mirko Viroli et al. "From distributed coordination to field calculus and aggregate computing". In: *Journal of Logical and Algebraic Methods in Programming* 109 (2019), p. 100486. ISSN: 2352-2208. DOI: https://doi.org/10.1016/j.jlamp.2019.100486. URL: https://www.sciencedirect.com/science/article/pii/S235222081930032X.

[13] Kev Zettler. *What is a distributed system?* URL: https://www.atlassian.com/microservices/microservices-architecture/distributed-architecture (visitato il 07/10/2024).

Acknowledgements

Optional. Max 1 page.