



Navigazione basata su inseguimento di frecce

Relazione di progetto

Progetto del corso Robotica (principi e progetto)

Università degli Studi di Bergamo

A.A. 2019/2020

CALEGARI ANDREA - 1041183

PAGANESSI ANDREA - 1040464

PIFFARI MICHELE - 1040658

January 10, 2020

Contents

1	Stato dell'arte	1
2	Camera	5
2.1	Scelta della camera	5
2.2	Come ottenere le immagini dalla camera?	6
2.3	Throttle	6
2.4	CPU consumption	6
2.5	Posizionamento camera	6
2.6	Calibrazione	6
3	Gestione delle maschere	9
3.1	HSV	9
3.2	Frecce o cerchi ?	9
3.3	Maschere	10
3.4	Erosione e dilatazione	10
4	Identificazione delle forme	11
4.1	Definizione del problema	11
4.2	Interpolazione	11
5	Dalle <i>pixel coordinates</i> alle <i>world coordinates</i>	13
5.1	Definizione del problema	13
5.2	Da <i>world coordinates</i> a <i>camera coordinates</i>	14
5.3	Da <i>camera coordinates</i> a <i>film coordinates</i>	15
5.4	Da <i>film coordinates</i> a <i>pixel coordinates</i>	16
5.5	Problema inverso	16
6	Matrici di rotazione	19
7	Conclusioni	23
8	Successive modifiche	25
8.1	Modifica dell'altezza della camera	25
8.2	Modifica dell'inclinazione della camera	25
8.3	Modifica del terreno su cui si trova il robot	25
9	Indicazioni per l'utilizzo della libreria	27
9.1	Installazione driver camera	27
9.2	Avvio del codice	27

List of Figures

1.1	Struttura del codice	2
1.2	Base robotica addetta alla movimentazione	3
1.3	Base verticale sulla quale andare ad inserire la camera	3
2.1	Camera utilizzata nello stato iniziale del progetto	5
2.2	Camera utilizzata nello step successivo	5
2.3	Interfaccia grafica durante la calibrazione	7
3.1	RGB vs HSV	9
3.2	Erosione e dilatazione	10
5.1	Schema concettuale delle diverse coordinate in gioco	13
5.2	Posizione del world frame	14
5.3	Posizionamento della camera su supporto metallico	15
5.4	Descrizione del problema	15
5.5	Descrizione dell'ultima trasformazione	16
5.6	Problema inverso	16
5.7	Piano del pavimento nel camera frame	17
6.1	Frames scelti	20
6.2	Frame della camera	21
6.3	Visione d'insieme dei frames scelti all'interno dell'area di lavoro	22
7.1	Esempio esplicativo del risultato finale dell'algoritmo	23

1

Stato dell'arte

Obbiettivo: andare a implementare sistema di *visual navigation* per la base robotica in figura 1.2 facendo uso del supporto in figura 1.3.

Per il progetto del corso di Robotica è stato implementato un algoritmo in C++ che, partendo da un'immagine di input, risulta essere in grado di individuare frecce (sfruttando la libreria OpenCV) composte dalla somma di due differenti forme: un rettangolo, di un certo colore, e un triangolo, di un secondo colore differente marcatori di un determinato colore di forma circolare , fornendo in output le coordinate dei centri di tali marcatori rispetto ad un sistema di riferimento. La scelta della forma dei marcatori è ricaduta sulle frecce poichè esse rappresentano *simboli* in grado di fornire una doppia informazione: *direzione e orientamento*, che rappresentano appunto i dati forniti in uscita dalla nostra libreria, la cui struttura a blocchi è rappresentata in figura 1.1.

L'algoritmo è stato sviluppato col presupposto di avere a disposizione un sistema visivo composto da una sola telecamera RGB; inoltre esso fornisce la possibilità di impostare l'angolo che la telecamera forma con l'asse orizzontale e l'altezza della camera stessa rispetto al terreno.

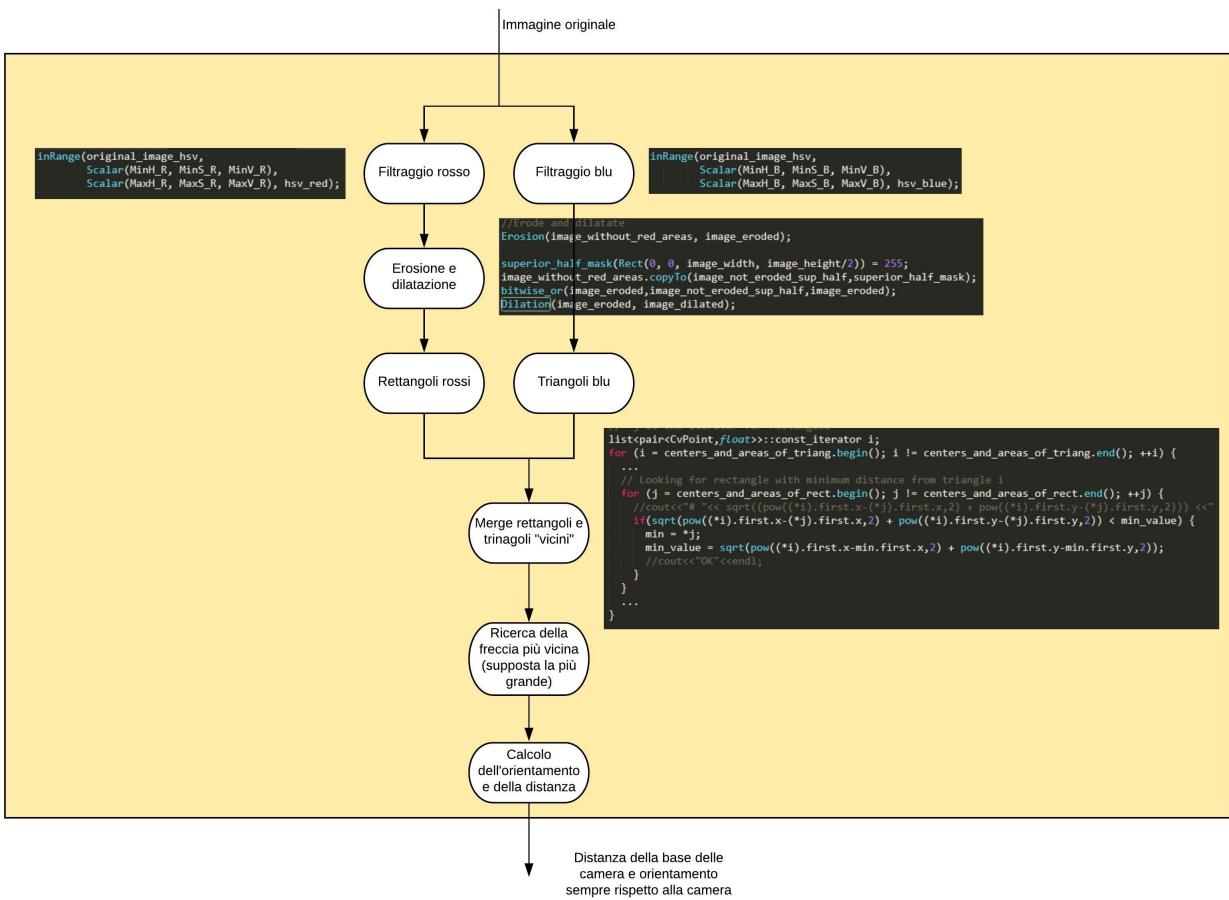


Figure 1.1: Struttura del codice



Figure 1.2: Base robotica addetta alla movimentazione



Figure 1.3: Base verticale sulla quale andare ad inserire la camera

2

Camera

2.1 Scelta della camera

Il progetto, allo stato iniziale, prevedeva di andare ad utilizzare la camera *SpotLight Pro Webcam* (figura 2.1), webcam già utilizzata in un progetto precedente, da cui abbiamo preso spunto per partire.



Figure 2.1: Camera utilizzata nello stato iniziale del progetto

È stato però notato che, utilizzando questa camera, non si era in grado di dare sufficienti garanzie di funzionamento stabile in alcune delle più comuni condizioni luminose e ambientali: infatti era alta la variabilità del comportamento della camera al variare delle condizioni luminose, il chè rendeva molto instabile il riconoscimento delle frecce.

Si è deciso quindi di passare ad una camera di tipo industriale, in grado di fornire delle prestazioni più stabili e affidabili.

La scelta è ricaduta sulla camera della casa produttrice *IDS (Imaging Development System)*: si tratta del modello *UI-1221LE-C-HQ* equipaggiata con la lente *BM2420* prodotta dalla *Lensagon* (lens datasheet).

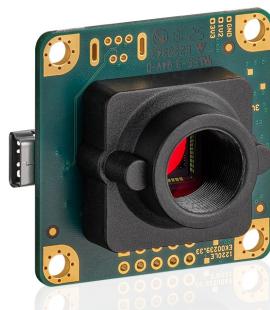


Figure 2.2: Camera utilizzata nello step successivo

La camera in figura 2.2 ha un'otturatore globale che permette di acquisire tutta l'immagine istantaneamente e non in modo progressivo come la camera *SpotLight Pro Webcam*.

2.2 Come ottenere le immagini dalla camera?

Per ottenere le immagini dalla camera è stato necessario iscriversi presso il sito web della casa produttrice e scaricare i driver necessari all'installazione e al funzionamento (software).

La videocamera ha diverse opzioni di acquisizione tra cui anche una grandezza dell'immagine diversa dal classico 640x480 pixel ma, per ragioni di semplicità, si è scelto di lasciare invariate le impostazioni di default.

In ogni caso è molto semplice verificare lo stato di funzionamento della camera stessa: è sufficiente, una volta scaricati e installati i software proprietari della casa produttrice, andare ad aprire il software *uEyeDemo* e modificare le impostazioni in base alle proprie necessità.

E' necessario sottolineare come, per essere in grado di ottenere le immagini dalla camera, sia importante assicurarsi che *l'ueye daemon* sia in funzione: esso parte automaticamente dal momento in cui si avvia il PC con la camera già connessa. Nel caso in cui essa venga connessa a caldo è necessario andare ad avviare il *daemon* utilizzando questo comando da terminale, che mette in evidenza come si tratti di un comando per camera usb (nel caso si andasse a lavorare con camera ethernet, servirebbe utilizzare ueyeethdrc):

```
1 sudo /etc/init.d/ueyeusbdrv start
```

2.3 Throttle

Durante la fase di test e verifica del codice si è scelto di utilizzare un throttle: esso non è altro che un topic predefinito di ROS il quale si occupa di prendere in input tutte le immagini provenienti dalla camera e regolare l'uscita in modo che l'output sia pubblicato con una frequenza minore dell' input; in questo modo si è ridotto sia il CPU consumption sia sia la variabilità dell'immagine in quanto le condizioni esterne sono soggette a rapidi ed imprevedibili cambiamenti. Si sono poi, ovviamente, svolti diversi test finali in condizioni di lavoro standard senza throttle.

2.4 CPU consumption

Un fattore importante nella scelta della camera è l'elevato tempo di utilizzo della CPU da parte della camera *UI-1221LE-C-HQ*. In contrasto, la camera inizialmente scelta vantava un consumo di CPU nettamente inferiore e quindi che più si potrebbe adattare all'installazione su dispositivi mobili e con bassa potenza di calcolo.

2.5 Posizionamento camera

Durante i test e gli esperimenti svolti in laboratorio la videocamera è stata legata ad un palo metallico e fissata con un angolo di inclinazione rispetto ad esso di circa 35 gradi; l'altezza dal pavimento è, inoltre, di circa 83.5 cm

2.6 Calibrazione

La fase di calibrazione della camera permette di ricavare (alcuni o tutti) i parametri che permettono al modello *pin-hole* di poter essere utilizzato per proiettare punti da coordinate mondo a coordinate camera.

In inglese la calibrazione della camera, ovvero il ricavare i parametri intrinseci e/o estrinseci, si chiama *Camera resectioning* in quanto il concetto di Camera Calibration si può riferire anche al problema della calibrazione fotometrica del sistema.

Una telecamera è infatti generalmente modellata mediante il modello proiettivo centrale pinhole camera (o foro stenopeico). Tale modello è definito da due famiglie di parametri:

- Parametri intrinseci (da tre a cinque): descrivono la telecamera indipendentemente dalla sua posizione nello spazio.
- Parametri estrinseci (sei): descrivono la posizione della telecamera nello spazio indipendentemente dalle sue caratteristiche interne.

Nel nostro caso specifico, per ottenere questi parametri di interesse, abbiamo sfruttato la forte interconnessione che la camera *IDS* offre con l'ambiente ROS: infatti, installando uno specifico pacchetto software tramite il comando

```
2 rosdep install camera_calibration
```

che permette di ottenere un insieme di pacchetti software scritti in *Python*, i quali permettono di ottenere tutti i parametri di interesse.

Nello specifico, il comando da eseguire tramite il terminale è il seguente:

```
3 rosrun camera_calibration cameracalibrator.py --size 8x6 --square 0.108 image:=/
  camera/image_raw camera:=/camera
```

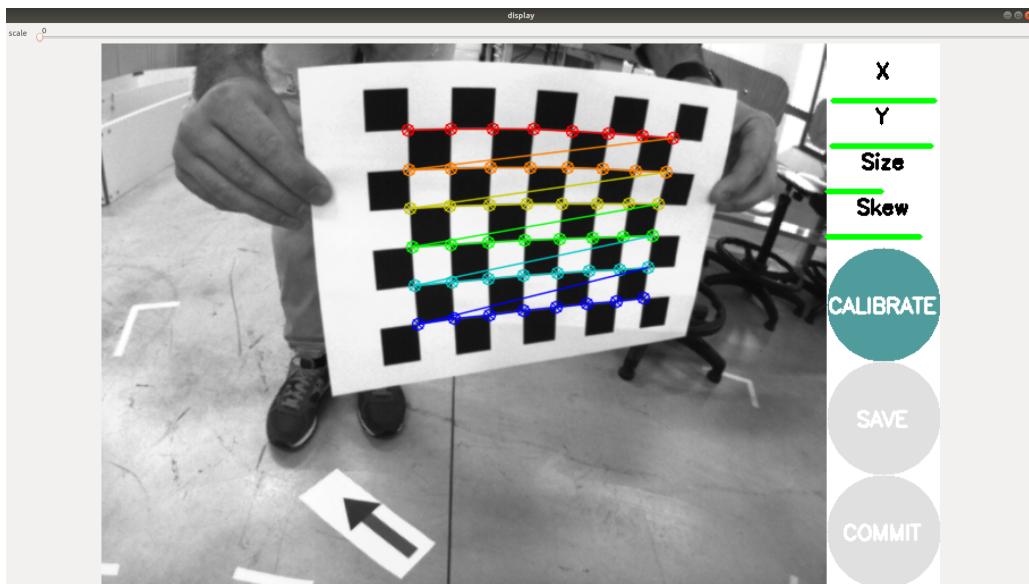


Figure 2.3: Interfaccia grafica durante la calibrazione

Questi invece sono gli esiti forniti in uscita dal processo di calibrazione:

```
4 width
5 640
6
7 height
8 480
9
10 [narrow_stereo]
11
12 camera_matrix
13 575.407063 0.000000 392.952424
14 0.000000 572.733663 256.805741
15 0.000000 0.000000 1.000000
16
17 distortion
```

```
18 -0.310832 0.140048 -0.005256 -0.007508 0.000000
19
20 rectification
21 1.000000 0.000000 0.000000
22 0.000000 1.000000 0.000000
23 0.000000 0.000000 1.000000
24
25
26
27 projection
28 517.612549 0.000000 400.977651 0.000000
29 0.000000 540.829285 256.819184 0.000000
30 0.000000 0.000000 1.000000 0.000000
```

3

Gestione delle maschere

3.1 HSV

Nella strutturazione del progetto ci è venuto molto naturale andare a lavorare con una scala di colori HSV.

Ma perchè non applicare un filtraggio basato su RGB?

Come noto nella letteratura, nell'ambito dell'*image recognition* è usuale il problema di andare a mascherare un colore piuttosto che un altro, come nel nostro caso.

Potremmo voler trovare, sempre per esempio, oggetti di colore rosso, scannerizzando nell'immagine solamente colori (255,0,0) nella scala RGB: con questo approccio andremmo ad applicare una condizione troppo stringente ai colori. Si potrebbe pensare, come soluzione a questa condizione parecchio stringente, di trovare colori in un range di rossi, come per esempio (130,0,0);(255,0,0): il problema comunque persisterebbe proprio per il fatto che il rosso è ottenuto come combinazione di più colori primari, e non come un solo singolo colore.

Potremmo pensare dunque, di andare a fondo del problema, cambiando gli intervalli dei valori RGB per tutti i colori primari ma sarebbe uno sforzo non secondario e probabilmente con risultati poco significativi. È necessario un metodo che abbia meno parametri per semplificare l'identificazione dei colori: questo metodo è rappresentato proprio dall'utilizzo del HSV.

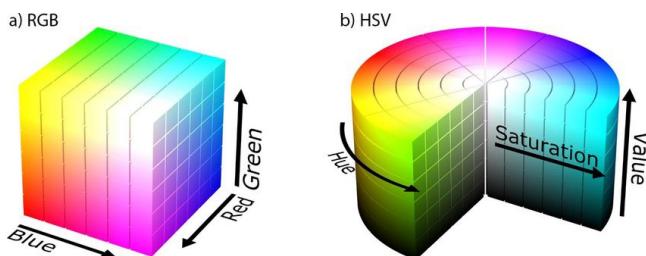


Figure 3.1: RGB vs HSV

Si può notare come, in Fig 3.1, il colore rosso sia definito in un range ben chiaro di un solo valore (*Hue*) mentre *Saturation* and *Value* servono solo a limitare ulteriormente la gamma di tonalità di rosso accettabili.

3.2 Frecce o cerchi ?

Nel progetto utilizzato come punto di partenza, si erano individuati cerchi di diversa grandezza per testare le funzionalità della libreria OpenCV nell'object detection.

Ovviamente l'utilizzo di cerchi limita di molto l'espressività del simbolo in quanto un cerchio non può, per sua natura identificare una direzione, men che meno un verso. Le frecce sono quindi la soluzione più naturale al problema di identificare con un simbolo una direzione ed un verso che poi, un eventuale robot mobile, potrà seguire.

TODO mettere foto cerchi e frecce foto maschera

3.3 Maschere

Le maschere sono fondamentali nel processo di object detection tramite la libreria OpenCV.

Queste ultime sono immagini binarie (bianco e nero) e sono utilizzate da tutte le funzioni di identificazione dei contorni e interpolazione di punti presenti in OpenCV. La seguente linea di codice permette di identificare tutti gli oggetti di colore rosso e di salvarli nella Matrice *hsv_red*.

```
31   inRange(original_image_hsv, Scalar(MinH_R, MinS_R, MinV_R), Scalar(MaxH_R, MaxS_R,
    MaxV_R), hsv_red);
```

3.4 Erosione e dilatazione

L'erosione e la dilatazione sono tecniche note allo stato dell'arte attuale utili per rimuovere rumore da una immagine.

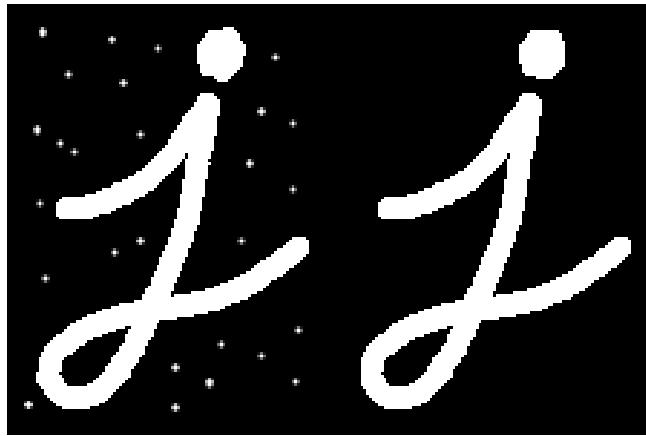


Figure 3.2: Erosione e dilatazione

Nel caso preso in considerazione da questo report è stato necessario applicare la tecnica di erosione solamente alla parte inferiore dell'immagine in quanto, se applicata nella parte superiore, avrebbe eliminato oltre al rumore anche un'eventuale freccia che sarebbe apparsa molto piccola.

Da sottolineare è il fatto che la tecnica di erosione e successiva dilatazione è stata applicata solamente alle figure rettangolari rosse in quanto la funzione di dilatazione prevede un kernel per la computazione del filtro: dato un kernel definito da una x e una y risulterebbe quanto mai complesso effettuare una dilatazione che rassomigli poi ad una forma triangolare.

4

Identificazione delle forme

4.1 Definizione del problema

Il capitolo precedente ha mostrato come sia possibile ottenere dei contorni delle figure da un'immagine a colori. È ora necessario identificare la forma di ciascuno di questi contorni riconoscendo così i vari quadrati e rettangoli presenti nell'immagine che avessero, una volta acquisiti dalla camera, il colore specificato. Come ultimo passaggio va svolto il controllo per verificare che esista o meno una freccia; quest'ultima altro non è che un triangolo e un rettangolo sufficientemente vicini fra di loro.

4.2 Interpolazione

Per approssimare il contorno ottenuto e filtrato attraverso le mask, come spiegato nel Capitolo 3, è necessario usare una funzione che approssima il contorno individuato con un altro poligono avente meno vertici così che la distanza tra di essi sia inferiore ad una certa soglia. Tale funzione è così definita nella libreria OpenCV:

```
32 void approxPolyDP(InputArray curve, OutputArray approxCurve, double epsilon, bool closed)
```

Si è reso necessario effettuare un tuning del parametro *epsilon* in quanto, per frecce diverse, a distanza variabile e con orientazione non fissa sono stati individuati differenti valori ottimali. Il valore che più si adattava a tutti i casi presi in considerazione è stato ottenuto sperimentalmente e corrisponde a *epsilon* = 0.045.

La funzione di cui sopra restituisce quindi una lista di poligoni ognuno dei quali è descritto da una lista di vertici. Il passo successivo è stato cercare nella lista dei poligoni un elemento che avesse 4 lati nel caso di un rettangolo e 3 in quello di un triangolo:

```
33 if(result->total >= 3 && result->total <= 3 && fabs(cvContourArea(result, CV_WHOLE_SEQ))>lower_area_triang)  
  
34 if(result->total >= 4 && result->total <= 6 && fabs(cvContourArea(result, CV_WHOLE_SEQ))>lower_area_rect)
```

Sempre per via sperimentale è stato possibile scoprire che vincolando il poligono che approssima il quadrilatero cercato ad avere tra i 4 e i 6 lati, la probabilità di riconoscere correttamente un rettangolo aumentava. Per il triangolo questo non si è reso necessario vista la buona probabilità di successo nella ricerca vincolata a 3 lati.

Ottenuti ora tutti i triangoli e i rettangoli sufficientemente grandi nella figura va affrontato il problema del riconoscimento di ogni freccia presente nel seguente modo:

- per ciascun rettangolo identificato, si calcola la distanza che intercorre tra esso e ogni triangolo riconosciuto. Per calcolare la distanza tra due figure è necessario:
 - definire il centro del rettangolo, tramite funzioni della libreria OpenCV;

- ottenere il baricentro del triangolo;
- calcolare la distanza cartesiana tra i due punti appena individuati.
- si tiene in considerazione solamente la distanza minore calcolata.
- si confronta suddetto valore con una soglia sperimentale; se questo valore è minore allora si può assumere che il triangolo e il quadrato presi in considerazione siano una freccia, altrimenti si scarta la coppia.
- la freccia appena rilevata viene aggiunta alla lista delle frecce rilevate nell'immagine.

Per ogni freccia, che ora altro non è che una coppia di punti,

$$\begin{aligned} C_{triangolo} &= (x_t, y_t) \\ C_{rettangolo} &= (x_r, y_r) \end{aligned} \quad (4.1)$$

vanno identificati nell'ordine:

- il centro della freccia, ottenuto come il punto medio del segmento che collega i due centri che definivano la freccia precedentemente.

$$C_{freccia} = \left(\frac{x_t + x_r}{2}, \frac{y_t + y_r}{2} \right)$$

- L'inclinazione della freccia nel piano, calcolata come:

$$\begin{aligned} \phi &= \text{atan}\left(\frac{\Delta y}{\Delta x}\right), \text{ dove} \\ \Delta y &= y_t - y_r \\ \Delta x &= x_t - x_r \end{aligned} \quad (4.2)$$

- L'area dell'oggetto freccia, ricavata come somma dell'area del triangolo e del quadrato.

5

Dalle *pixel coordinates* alle *world coordinates*

5.1 Definizione del problema

Nel capitolo precedente è stato illustrato un metodo atto all'identificazione delle frecce che rientrano nel campo visivo della camera. Viene ora trattato come sia possibile ottenere la posizione della freccia, precedentemente identificata, nelle coordinate 3D (U,V,W) rispetto alla base del robot.

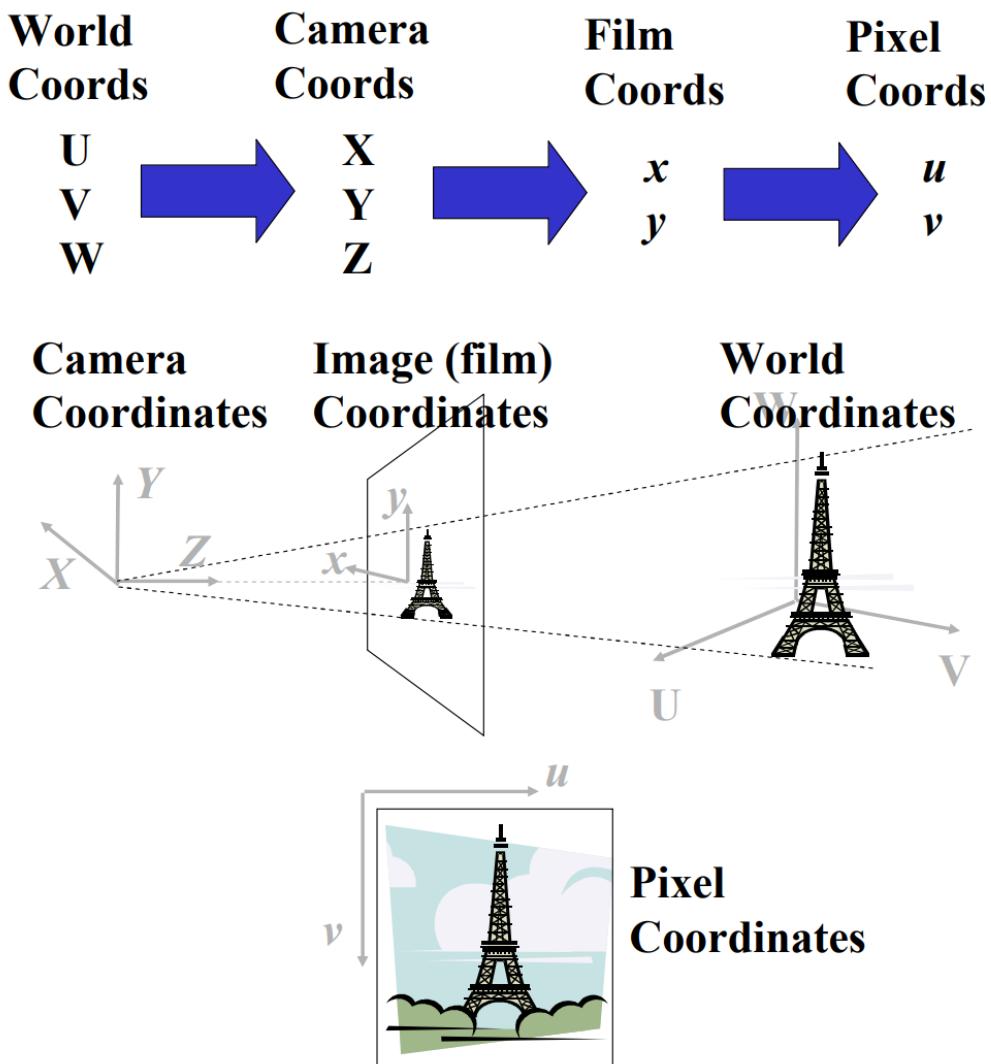


Figure 5.1: Schema concettuale delle diverse coordinate in gioco

Come si vede in figura 5.1, sono necessarie tre trasformazioni per ottenere, a partire dalle *world coordinates*, le *pixel coordinates*.

Nel caso specifico del sistema preso in considerazione all'interno di questo report, il problema risulta essere l'opposto: dalle coordinate nella camera è necessario ottenere la posizione globale dell'oggetto effettuando una trasformazione inversa.

Si analizzeranno ora le singole trasformazioni che permetteranno alla fine di ottenere il risultato voluto.

5.2 Da *world coordinates* a *camera coordinates*



Figure 5.2: Posizione del world frame

Partendo da un sistema di riferimento solidale al robot e posto all'altezza del pavimento, che identifichiamo come sistema globale, è possibile definire una matrice di rototraslazione per ottenere il sistema di riferimento solidale al centro della camera.

$$R_{worldcam} = R_{traslazione} \cdot R_{rotazione} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & h_{cam} \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\frac{\pi}{2} + cam_incl) & -\sin(\frac{\pi}{2} + cam_incl) & 0 \\ 0 & \sin(cam_incl + \frac{\pi}{2}) & \cos(cam_incl + \frac{\pi}{2}) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

¹

È stata effettuata una traslazione lungo l'asse W in quanto la camera è posta esattamente sopra l'origine del sistema O_{UVW} ed una rotazione rispetto all'asse U di $\frac{\pi}{2}$ in quanto, per convenzione, si associa all'asse delle Z la profondità nel frame solidale alla camera.

A questo punto è necessario effettuare un'altra rotazione di *cam inclination* gradi rispetto all'asse X a seconda dell'inclinazione alla quale si sceglie di far lavorare la camera, come si vede in figura 5.3.

¹ *cam incl* corrisponde, all'interno del codice, alla variabile *cam inclination*

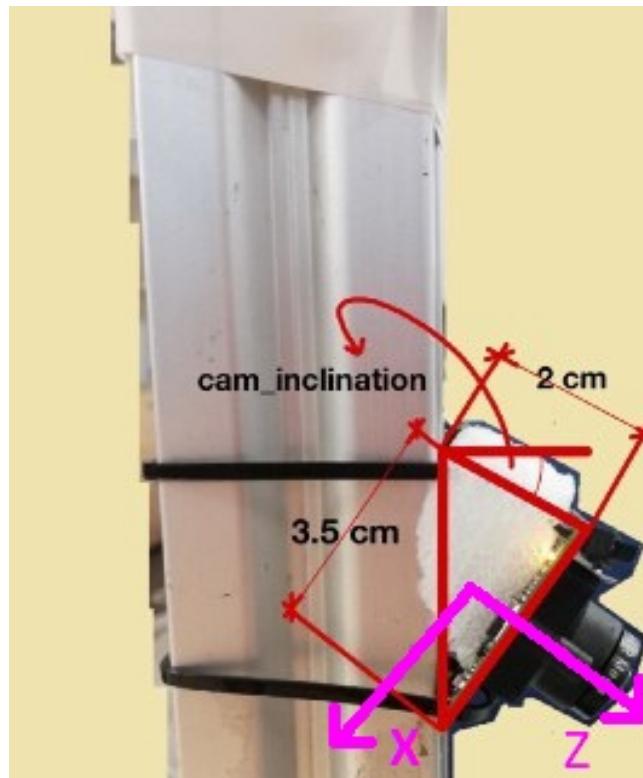


Figure 5.3: Posizionamento della camera su supporto metallico

5.3 Da camera coordinates a film coordinates

Basic Perspective Projection

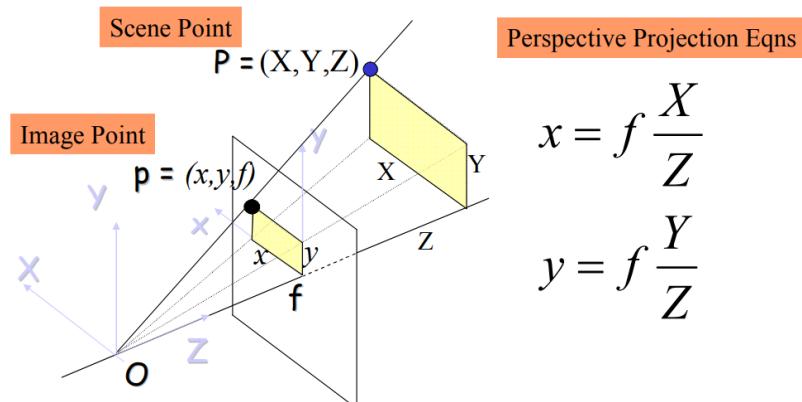
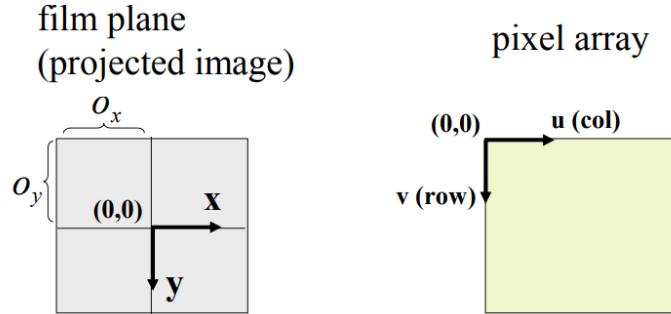


Figure 5.4: Descrizione del problema

Nello schema in figura 5.4, f rappresenta il fuoco della camera: si tratta di un parametro ottenibile attraverso la procedura di calibrazione, il quale può, come il centro della camera, essere scomposto in due componenti f_x e f_y , utilizzati rispettivamente, per trasformazioni lungo l'asse x e lungo l'asse y .

5.4 Da *film coordinates* a *pixel coordinates*

Intrinsic parameters (offsets)



$$u = f \frac{X}{Z} + o_x \quad v = f \frac{Y}{Z} + o_y$$

Figure 5.5: Descrizione dell'ultima trasformazione

In figura 5.5, i termini O_x e O_y rappresentano i centri della camera e sono anch'essi ricavabili tramite la procedura di calibrazione della camera.

5.5 Problema inverso

Backward Projection

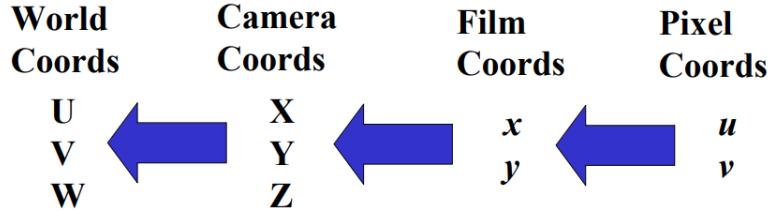


Figure 5.6: Problema inverso

Come si vede in figura 5.6, si deve ora affrontare il processo inverso siccome, nel nostro caso, si hanno a disposizione u e v e si vogliono ottenere U, V, W .

Come già detto in precedenza, O_x e O_y e f sono ottenibili tramite calibrazione. e quindi:

$$\begin{aligned} x &= u - O_x \\ y &= v - O_y \end{aligned} \tag{5.1}$$

In questo modo sono quindi state ottenute le equazioni del film coordinates.

È ora necessario ottenere le camera coordinates. Per fare ciò è necessario conoscere il valore di Z (ovvero la profondità), la quale è ottenibile in due modi:

- utilizzando una camera con sensore di profondità

- assumendo che gli oggetti inquadrati dalla camera siano sempre posti su un piano di cui si conosce l'equazione.

La seconda assunzione è, nella realtà dei fatti, un'ipotesi corretta e applicabile in quanto, nel nostro caso, gli oggetti e le frecce giaceranno sempre sul pavimento.

È quindi richiesto di calcolare l'equazione del pavimento nel camera frame:

- $z = 0$ rappresenta l'equazione del piano se fosse nel world frame
- $z \cdot R_{worldcam} = 0$ indica il piano così calcolato è la descrizione dal punto di vista matematico del pavimento dal punto di vista della camera

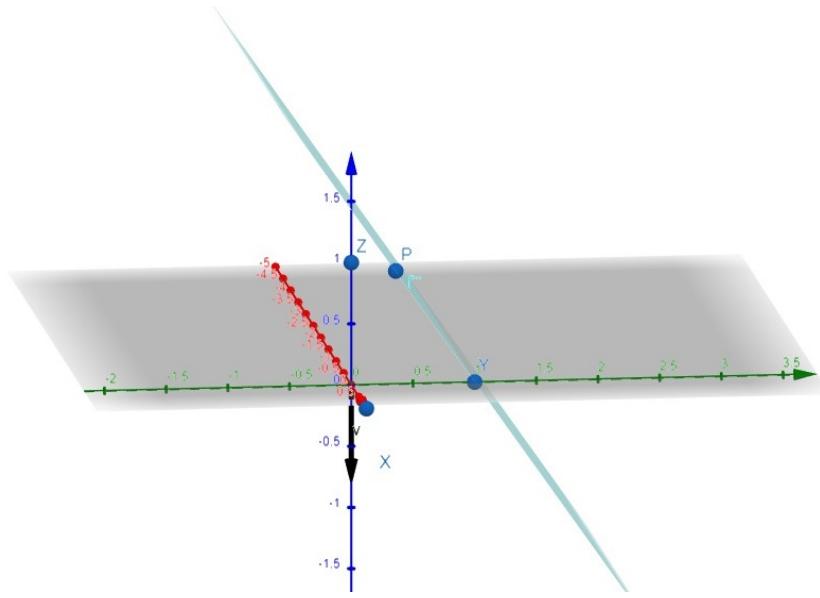


Figure 5.7: Piano del pavimento nel camera frame

- si può calcolare il fattore di scala s dato un generico piano $ax + by + cz + d = 0$

$$s = \frac{-d}{aX' + bY' + c} \quad (5.2)$$

dove

$$\begin{aligned} X' &= \frac{x}{f_x} = \frac{X}{Z} \\ Y' &= \frac{y}{f_y} = \frac{Y}{Z} \end{aligned} \quad (5.3)$$

che sono le coordinate normalizzate rispetto a Z .

- Dunque, come ultimo passaggio si moltiplica tutto per il fattore di scala:

$$\begin{aligned} X &= X' \cdot s \\ Y &= Y' \cdot s \\ Z &= S \end{aligned} \quad (5.4)$$

Per ottenere le equazioni nel world frame si deve dunque utilizzare la matrice inversa:

$$\begin{pmatrix} U \\ V \\ W \\ 1 \end{pmatrix} = R_{worldcam}^{-1} \cdot \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

6

Matrici di rotazione

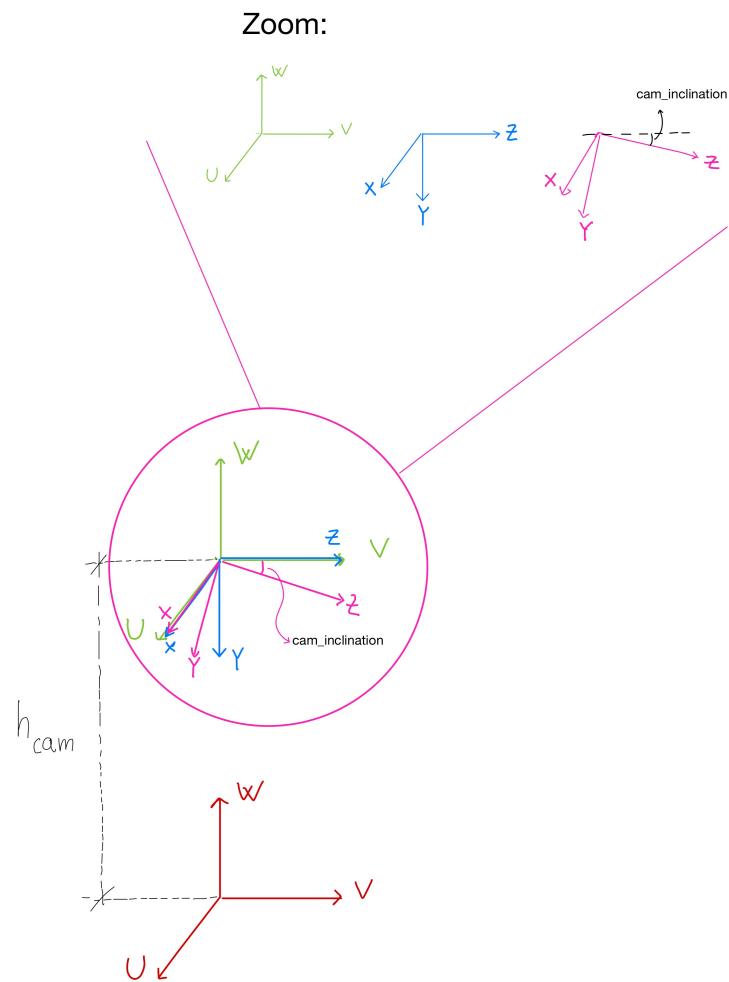


Figure 6.1: Frames scelti

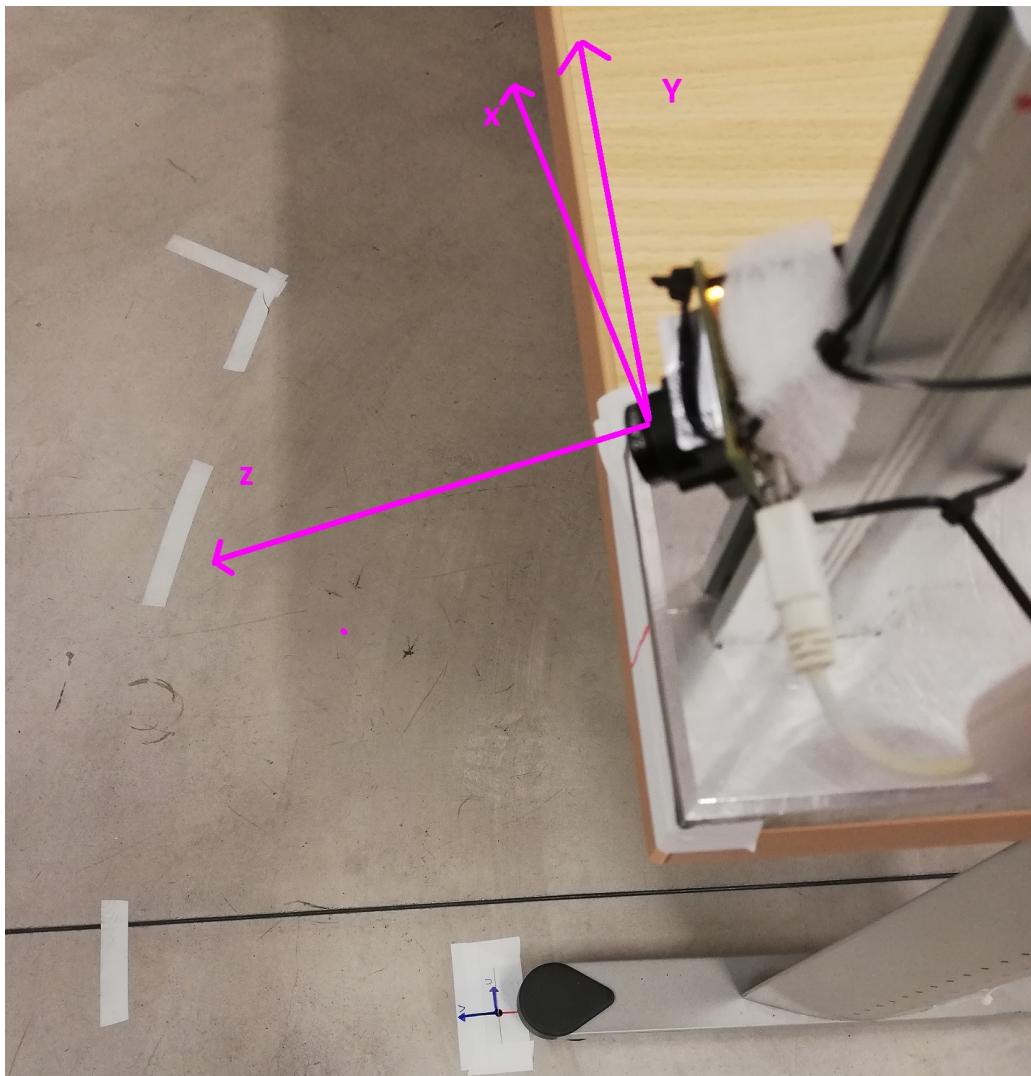


Figure 6.2: Frame della camera

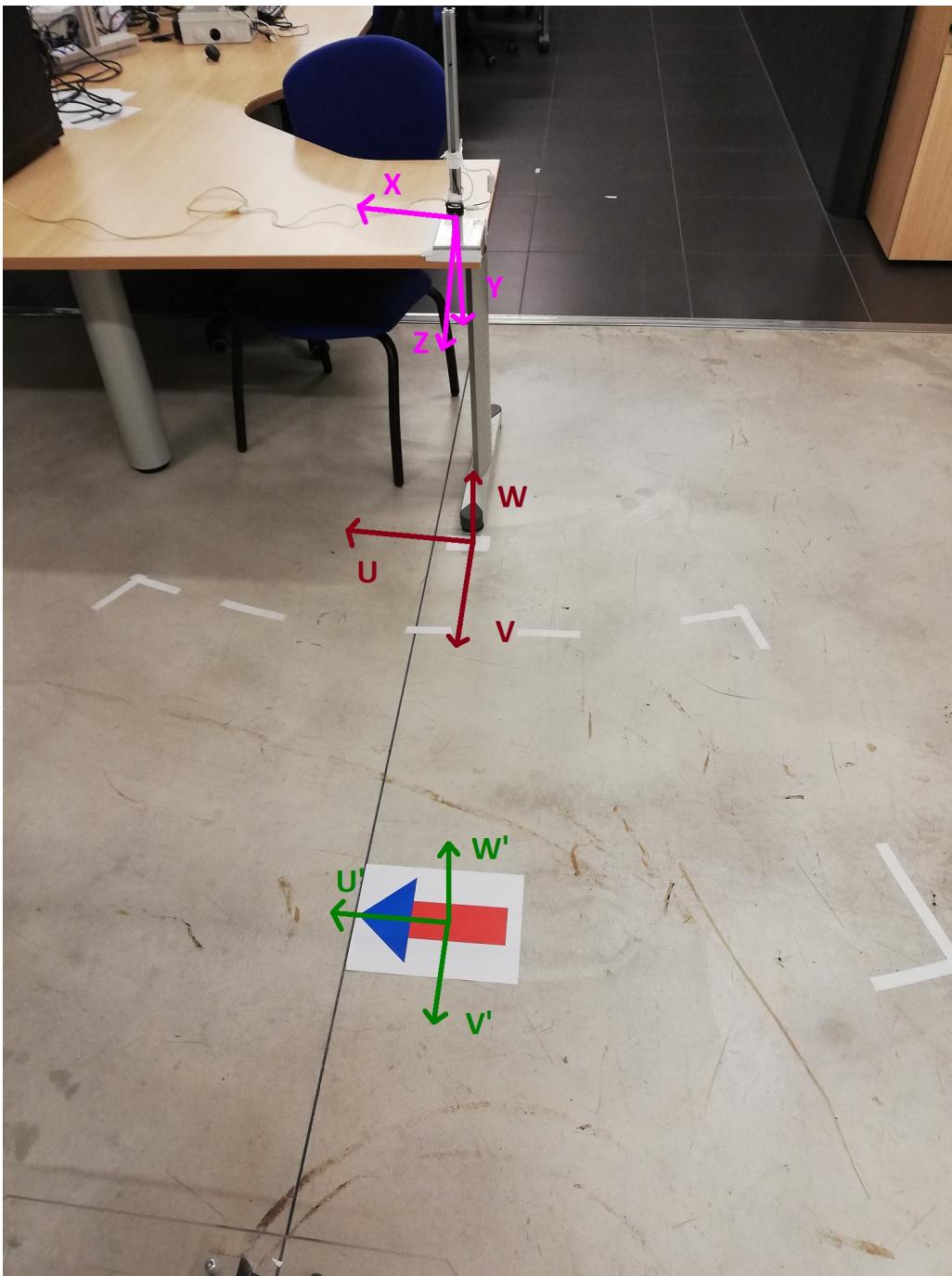


Figure 6.3: Visione d'insieme dei frames scelti all'interno dell'area di lavoro

Le seguenti sono le matrici utilizzate nel codice per rappresentare le rotazioni dei sistemi di riferimento.

```

35 R_traslation << 1,0,0,0,
36     0,1,0,0,
37     0,0,1,-h_cam,
38     0,0,0,1;
39
40 R_rot_cam_inclination << 1,0,0,0,
41     0,cos(cam_inclination),-sin(cam_inclination),0,
42     0,sin(cam_inclination),cos(cam_inclination),0,
43     0,0,0,1;
44
45 R_rot_camera << 1,0,0,0,
46     0,cos(M_PI/2),-sin(M_PI/2),0,
47     0,sin(M_PI/2),cos(M_PI/2),0,
48     0,0,0,1;

```

7

Conclusioni

TODO: DUE COMMENTI (come documentazione vecchio progetto)

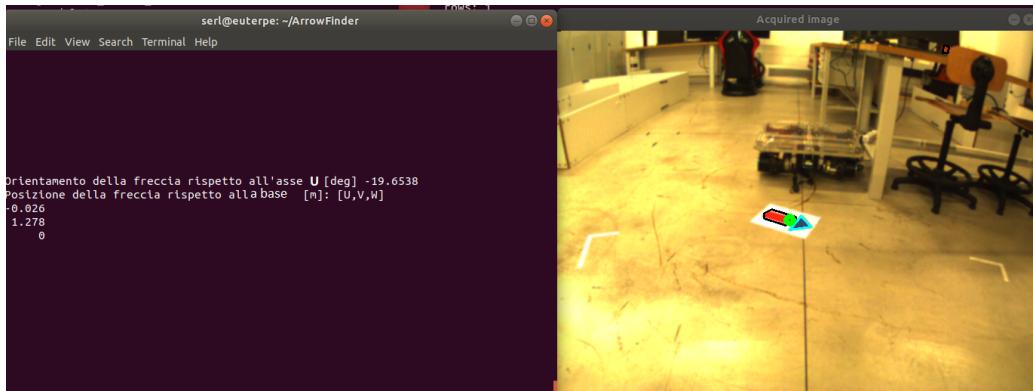


Figure 7.1: Esempio esplicativo del risultato finale dell'algoritmo

8

Successive modifiche

8.1 Modifica dell'altezza della camera

Per effettuare modifiche all'altezza della camera è sufficiente modificare il parametro h_{cam} della matrice:

$$R_{traslazione} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -h_{cam} \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (8.1)$$

Nel caso di modifiche anche rispetto ad altri assi quali U e V sarà sufficiente modificare i valori dei primi due elementi della quarta colonna che rappresentano rispettivamente traslazioni lungo l'asse U e l'asse V.

8.2 Modifica dell'inclinazione della camera

Per effettuare eventuali modifiche alla inclinazione della camera, sempre ammesso che il piano Z si voglia uscente dal piano della camera, sarà allora sufficiente modificare il parametro α , che appare nella matrice seguente:

$$R_{rotazione} = \begin{pmatrix} \cos(\frac{\pi}{2} + \alpha) & -\sin(\frac{\pi}{2} + \alpha) & 0 & 0 \\ \sin(\frac{\pi}{2} + \alpha) & \cos(\frac{\pi}{2} + \alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (8.2)$$

8.3 Modifica del terreno su cui si trova il robot

Nel caso in cui il robot su cui è montata la camera dovesse essere utilizzato in ambienti diversi da quelli di un laboratorio in cui sono assenti salite, discese o dislivelli allora sarà necessario modificare il piano su cui la freccia giace.

Per fare ciò sarà necessario ricalcolare il piano su cui si trova la freccia $ax + by + cz + d = 0$ e ricalcolare il fattore di scala

$$s = \frac{-d}{aX' + bY' + c} \quad (8.3)$$

Si nota come, se il robot dovesse viaggiare su un pavimento senza variazioni lungo l'asse U, allora il fattore di scala sarebbe definito come:

$$s = \left| \frac{h_{cam}}{-\sin(\frac{\pi}{2} + \alpha) \cdot Y' + \cos(\frac{\pi}{2} + \alpha)} \right|$$

9

Indicazioni per l'utilizzo della libreria

Andiamo qui a riassumere alcuni passi operativi che sono stati realizzati durante lo sviluppo ed il proseguimento del progetto.

9.1 Installazione driver camera

TODO - download e installazione driver

Nel caso in cui la camera sembri non funzionare, come specificato nel capitolo 2.2, è necessario avviare il daemon della camera. Questo è possibile solo se esso non è già stato attivato come nel caso, per esempio, di un plug-in a caldo. Il comando da runnare da terminale, in qualsiasi *posizione*, è il seguente:

```
49 - sudo /etc/init.d/ueyeusbdrv start  
50 (il comando iniziale permetteva di scegliere tra usb e eth: la nostra camera e usb)  
!
```

9.2 Avvio del codice

Nella nostra esperienza ci è stato molto comodo andare ad utilizzare non il classico terminale messo a disposizione da Ubuntu, ma bensì ci siamo appoggiati all'utilizzo di *terminator*, il quale permette una migliore gestione di terminali multipli.

Ecco una sequenza di comandi da inserire da terminale per poter andare ad eseguire il codice.

```
51  
52 Scrivere su tutti terminali (e su tutti quelli che verranno aperti) i seguenti  
53 comandi per il \textit{bash} del progetto:  
54 - CMD_1: cd ArrowFinder/devel/; . setup.bash  
55 - CMD_2: cd ..  
56  
57 Terminale 1:  
58 - Caricare il bash da devel (CMD_1 / CMD_2)  
59 - CMD_3: roscore  
60  
61 Terminale 2:  
62 - Caricare il bash da devel (CMD_1 / CMD_2)  
63 - CMD_4: rosrun arrow_finder arrow_finder_node  
64  
65 Terminale 3:  
66 - Caricare il bash da devel (CMD_1 / CMD_2)  
67 - CMD_5: cd ~/ArrowFinder/src/ueye_cam/launch  
68 - CMD_6: roslaunch debug.launch  
69  
70 Terminale 4:  
 - CMD_7: rosrun topic_tools throttle messages /camera/image_raw 5.0
```

Da evidenziare come, il comando eseguito sul terminale 4, risulta essere utile nel caso in cui sia necessario andare a temporizzare, in maniera automatica e gestita completamente da ROS, la lettura dell'immagine

Bibliography

- [1] *Descrizione della camera* <https://en.ids-imaging.com/store/ui-1221le-rev-2.html>
- [2] *Manuale della camera* https://en.ids-imaging.com/IDS/datasheet_pdf.php?sku=AB02422
- [3] *Manuale della lente* <https://www.lensation.de/product/BM2420/>
- [4] *Ueye cam e ROS* <http://wiki.ros.org/ueye>
- [5] *HSV vs RGB* <https://medium.com/neurosapiens/segmentation-and-classification-with-hsv-8f2406d0a2>
- [6] *HSV vs RGB* <https://handmap.github.io/hsv-vs-rgb/>
- [7] *Camera Projection I* <http://www.cse.psu.edu/~rtc12/CSE486/lecture12.pdf>
- [8] *Camera Projection II* <http://www.cse.psu.edu/~rtc12/CSE486/lecture13.pdf>
- [9] *Coordinate omogenee* http://robotics.unibg.it/teaching/robotics/pdf/14_Geometria3D.pdf
- [10] *Calibrazione camera I* <http://www.ce.unipr.it/people/medici/geometry/node145.html>
- [11] *Calibrazione camera II* http://wiki.ros.org/camera_calibration