Universitá di Trento

Dipartimento di Ingegneria e Scienza dell'Informazione

Corso di Laurea in Informatica

Tesi Finale

RICONOSCIMENTO E STIMA DISTANZA DI CARTELLI STRADALI TRAMITE WEBCAM

Relatore:
Prof. Luigi Palopoli

Candidato:
Luca Zamboni

Correlatore:
Federico Moro

Anno academico 2013-2014

Ai miei genitori
A mio fratello

Indice

Elenco delle figure

Capitolo 1

Introduzione

1.1 Motivazioni

Negli ultimi anni si è sviluppata la necessità di realizzare robot o macchine in grado di analizzare e muoversi autonomamente nell' ambiente senza bisogno dell' aiuto umano. Un settore importante in questo campo è la computer vision o visione artificiale. Per computer vision si intendono l'insieme dei processi che mirano a riprodurre la visione umana allo scopo di percepire, analizzare e riconoscere l'ambiente circostante. Negli ultimi tempi la computer vision si è sviluppata rapidamente in molti settori, basti pensare alle applicazioni di sicurezza delle automobili dove la visione della strada permette di rilevare eventuali pericoli sul terreno e di conseguenza effettuare frenate di emergenza. In un prossimo futuro l'integrazione fra computer vision, sia nello spettro di luce visibile che nell'infrarosso, ultravioletto, raggi-x, sensoristica di controllo auto e un sistema gps permetterà il completo controllo della viabilità stradale.

1.2 Il progetto

Lo scopo del progetto è quello di far muovere un robot lungo un percorso contrassegnato da una linea nera sul terreno. Il robot deve seguire la linea nera avvalendosi di due telecamere: una laterale ed una frontale. Entrambe le telecamere sono collegate a due schede Beaglebone[1], le quali comunicano tra di loro e che servono per l' elaborazione delle immagini. Le immagini catturate dalla telecamera frontale sono utilizzate per riconoscere il percorso dettato dalla linea nera e riuscire a fare una pianificazione di una traiettoria, invece

[1] `http://beagleboard.org/bone`

la telecamera laterale è utlizzata per sapere a che distanza si trova la linea e corregge la traiettoria del robot. Il robot sa a priori la struttura del percorso e deve riuscire a localizzarsi su di esso. Per sapere dov'è la sua posizione utilizza degli encoder posti sulle ruote. In questo modo si sa esattamente quanti giri ha fatto ciascuna ruota e si riesce a calcolare facilmente la posizione del dispositivo. Purtroppo però ci sono sempre degli errori collegati agli encoder, anche se piccoli, questi si accumulano nel tempo rendendo la stima della posizione errata o non completamente utilizzabile. Di conseguenza è stato deciso di inserire nel percorso dei cartelli in posizioni note e stimando la posizione e la distanza da essi si riesce a correggere i valori relativi agli encoder.

1.3 Obbiettivi

In questo progetto per il riconoscimento dei cartelli posizionati sul percorso si parte da un algoritmo già realizzato e si propone un modo per migliorarne le prestazioni nella velocità dell' elaborazione delle immagini, aumentare la precisione e la distanza a cui viene riconosciuto un segnale e infine stimare la distanza da esso.

L' algoritmo di riconoscimento cartelli prende in input le immagini dallo streaming video della telecamera frontale. Esso deve elaborare ogni frame e riuscire a individuare se in essa è presente un cartello e in seguito riuscire a riconoscere di che tipo di cartello si tratti. Dal momento che l'algoritmo deve essere eseguito su un corpo in movimento questo impone che il calcolo debba avere un tempo di esecuzione abbastanza ristretto. Facendo un esempio: se il robot si muovesse alla velocità di 10 Km/h esso percorrerebbe ~3 m/s (2.7777 m/s). Quindi se l'algoritmo per riconoscere il cartello impiegasse un secondo, significherebbe che se un cartello è posizionato a meno di 3 metri allora viene riconosciuto solo quando è stato superato. Dunque si cerca di contrarre il più possibile il tempo di esecuzione dell' algoritmo. Oltre a questo bisogna che il riconoscimento del cartello sia robusto. Per robustezza si intende la capacità di:

- Riconoscere il segnale a varie distanze sia da vicino che da lontano.
- Individuare il cartello con diversi gradi di luminosità. Questo perchè il robot deve essere in grado di guidare autonomamente, sia quando c'è una giornata di sole che quando c'è una giornata nuvolosa.
- Riuscire a riconoscere il segnale da varie angolature, attuando una trasformazione prospettica. Questo perchè mentre si è in moto non sempre si è perfettamente allineati con il cartello.

- Riuscire a filtrare le forme geometriche dei cartelli, tralasciando le altre. Se ad esempio in un'immagine sono presenti più quadrati, individuare tra questi solo quelli relativi ai cartelli.

1.4 Outline

Dopo questa breve introduzione, la tesi si struttura su quattro capitoli.

- Capitolo 2: offre una descrizione di come opera l'algoritmo esistente e vengono analizzate e discusse le sue prestazioni e la sua efficacia.
- Capitolo 3: sono proposte soluzioni per migliorare l'algoritmo descritto nel capitolo precedente.
- Capitolo 4: vengono esposti e analizzati i dati raccolti dalle prove sperimentali a seguito dei nuovi cambiamenti introdotti.
- Capitolo 5: conclusioni del lavoro svolto.

Capitolo 2

Analisi dell' algoritmo esistente

In questo capitolo verrà analizzato il codice dell' algoritmo esistente e verrano valutate prestazioni ed efficacia di esso.

2.1 Ambiente di sviluppo

L' hardware utilizzato per il progetto é il seguente:

- Processore : AM335x 720MHz ARM Cortex-A8
- RAM: 256 MB
- Telecamera: WebCam 640 x 480 pixel

L' algoritmo per riconoscere i segnali ha la necessità di elaborare le immagini provenienti da uno stream video. Essendo che il codice viene eseguito su una scheda con processore non particolarmente potente e avendo a disposizione poca RAM, le performance per questo algoritmo sono un aspetto fondamentale. Per questo motivo è stato scelto come linguaggio c++ che offre ottime prestazioni. Inoltre viene utilizzata la libreria open source chiamata OpenCV che offre molte funzioni per l' elaborazione immagini e video ed è molto indicata per applicazioni in real time. Inoltre, OpenCV non impone nessuna restrizione sulla licenza del software.[1]

2.2 I cartelli

I cartelli individuati da questo algoritmo sono quelli rappresentati nelle figure 2.1. Essi hanno un contorno nero su sfondo bianco con all' interno una figura

[1] `http://opencv.org`

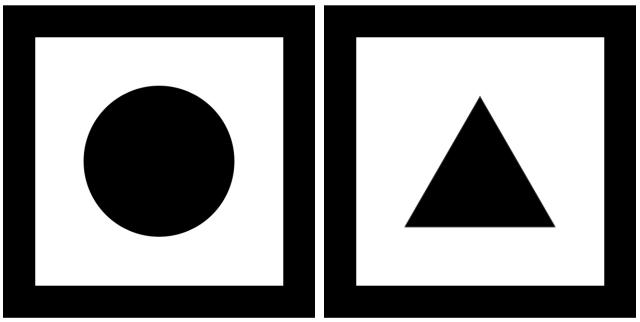

Figura 2.1: Cartelli riconosciuti

che può essere un triangolo o un cerchio. É stato scelto di usare un bordo nero per facilitare il riconoscimento del cartello, in quanto se l'immagine che lo contiene avesse del bianco attorno al simbolo ed anch' esso fosse bianco allora sarebbe impossibile o molto difficile poterlo individuare. Invece, così facendo, si ha uno stacco netto con le figure sul retro del cartello. La larghezza della parte bianca del cartello ha una proporzione di $\frac{8}{5}$ rispetto al diametro del cerchio e alla base del triangolo.

2.3 Impostazione generale

Passiamo adesso all' analisi dell' algoritmo esistente descrivendo i principali passi di esecuzione che verranno analizzati in dettaglio successivamente. L'algoritmo prende in input un' immagine presa dallo streaming video della telecamera frontale. L'immagine è rappresentata con una matrice di pixel N x M (nel progetto tipicamente 640x480 pixel). La cella [i,j] contiene il colore del pixel corrispondente. Ad alto livello l'algoritmo esegue le seguenti operazioni:

- Individuazione dei cartelli nell' immagine nel nostro caso dei quadrati chiamati ROI (Region of Interest).
- Esecuzione di una trasformazione prospettica delle ROI che porta alla rimozione dell' angolatura facendo risultare l'immagine come se fosse visualizzata frontalmente.
- Individuazione di un triangolo o di un cerchio all'interno dei quadrati cioè riconoscere il cartello scartando presunte figure non tali.

Di seguito passeremo ad analizzare nel dettaglio i passi dell'algoritmo.

2.3.1 Individuazione cartello

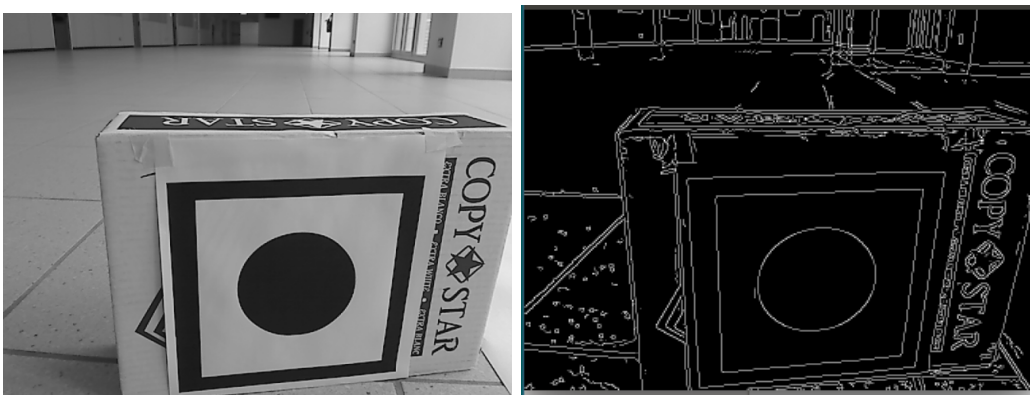

Figura 2.2: A sinistra l' immagine originale, a destra l'immagine a cui è stato applicato il filtro di Canny.

Prima di procedere all' effettiva individuazione del cartello è necessario trasformare l'immagine proveniente dallo stream video in bianco e nero. Per fare tale operazione si utilizza la funzione di OpenCV cv:cvtColor(). Successivamente all' immagine in scala di grigi si applica il filtro di Canny [1]. Il filtro di Canny serve a individuare i bordi degli oggetti in un' immagine. L'algoritmo di Canny si basa sul seguente metodo: si passano due valori alla funzione che saranno poi le nostre soglie. Poi si passa ad analizzare l'immagine punto per punto calcolandone il gradiente in esso. Se il gradiente è:

- Inferiore alla soglia bassa, il punto è scartato;
- Superiore alla soglia alta, il punto è accettato come parte di un contorno;
- Compreso fra le due soglie, il punto è accettato solamente se contiguo ad un punto già precedentemente accettato.

Per eseguire il filtro di Canny all' immagine su utilizza l'omonima funzione cv::Canny() di OpenCV. Una volta ottenuti i bordi, essi si dilatano con la funzione cv::dilate() per eliminare eventuali buchi ai vertici. Poi si utilizza la funzione cv::approxPolyDP() che prende in input i bordi e ricava i poligoni approssimati. Da questi poi si tengono solo i poligoni che hanno 4 lati e che sono convessi. Dopo di che, si applica un filtro il quale controlla che la differenza dei lati opposti abbia al massimo un certo errore cioè che siano di lunghezza simile e si eliminano anche eventuali duplicati. I quadrati risultanti sono le nostre ROI (Region of interest) cioè i nostri presunti cartelli dove andare a cercare il simbolo all' interno.

2.3.2 Trasformazione prospettica

Dopo aver trovato i presunti cartelli bisogna eseguire una trasformazione prospettica. Essa consiste nel portare il contenuto della ROI a una posizione frontale alla telecamera. Questo rendere accurata la fase successiva cioè riconoscere la forma all' interno del cartello. Per effettuare la trasformazione si usa la funzione cv::getPerspectiveTransform() che calcola la matrice di trasformazione del sistema. Poi viene usata la funzione cv::warpPerspective() che effettua l'effettiva trasformazione.

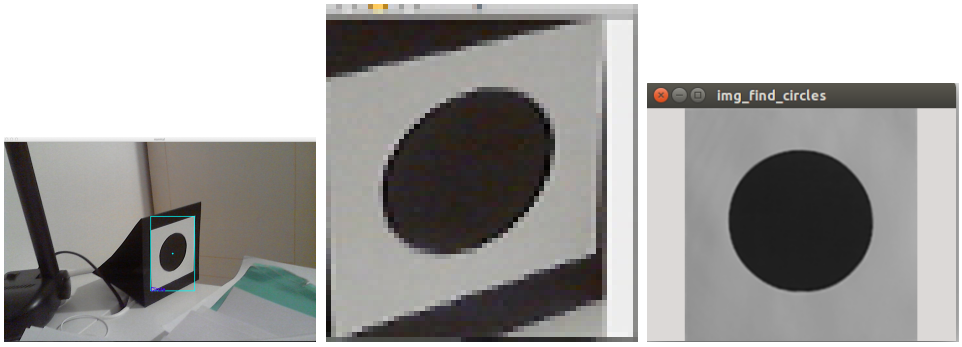

Figura 2.3: Trasformazione prospetica cartello

2.3.3 Riconoscimento cartello

Come ultimo passo bisogna determinare di che tipo di cartello si tratta oppure scartarlo, nel caso in cui esso era un falso cartello cioè una forma geometrica simile al cartello e riconosciuto come tale dai punti precedenti dell' algoritmo. Questo passo prende in input le ROI riconosciute ed elaborate in precedenza. Per riuscire a riconoscere i cerchi e i triangoli si usa una tecnica simile a quella per ricavare i quadrati dalla foto originale. In entrambe si usa il filtro di Canny per ricavare i bordi della figura. Una volta ricavati i bordi le strade si dividono per le due forme. Per riconoscere il triangolo si usa la funzione cv::approxPolyDP() come per riconoscere i quadrati peró in questo caso ovviamente si ricavano solo i poligoni con 3 lati. Invece per riconoscere il cerchio si usa la funzione cv::HoughCircle(). Questa funzione è contenuta nella libreria OpenCV e prende semplicemente in input l'immagine e restituisce i contorni che formano un cerchio entro un certo errore. Dopodichè, per entrambi i casi si controlla che la figura sia circa al centro del quadrato del cartello entro un certo grado di errore, altrimenti si scarta e si controlla che essa abbia le proporzioni giuste cioè la larghezza del cartello è $\frac{8}{5}$ della larghezza della figura interna anche qui entro un certo errore.

Arrivati a questo punto si è riusciti a determinare il tipo di cartello a cui appartiene la ROI passata in input.

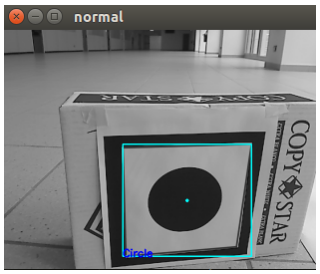

Figura 2.4: Cartello riconosciuto

2.4 Esperimenti eseguiti

Adesso passiamo all' analisi prestazioni e precisione algoritmo.

Per prima cosa analizziamo la precisione dell'algoritmo con immagini prese a varie distanze, con angolature non superiori ai 15° e di giorno cioè con luce accettabile. Questi sono i dati ricavati utilizzando cartelli con larghezza circa di un lato corto di un foglio A4 posti tra i 1 e 4 metri dalla posizione della telecamera.

	ROI riconosciuti	Segnali riconosciuti	Riconosciuti sbagliati
0 metri	100 %	100 %	0 %
1 metro	100 %	80 %	10 %
2 metri	100 %	10 %	25 %
3 metri	60 %	0 %	20 %
4 metri	20 %	0 %	0 %

Tabella 2.1: Percentuale successo riconoscimento cartelli

Da una prima analisi si evince che l'algoritmo ha delle difficoltà nel riconoscimento dovute alla distanza. Quello che si nota analizzando i dati della tabella è che l'algoritmo trova delle difficoltà a trovare ROI nell'immagine quando il cartello è posto a più di due metri di distanza. Inoltre, ha difficoltà ancora maggiori quando ha trovato il cartello ma deve determinare di che tipo di segnale si tratta. Infatti anche in questo caso se si va oltre il metro la precisione cala drasticamente. Di seguito analizzeremo i dati relativi ai tempi di esecuzione.

Analizzando il grafico si nota che l'algoritmo impiega mediamente 660 millisecondi. Facendo ulteriori analisi sui tempi delle singoli funzioni si nota

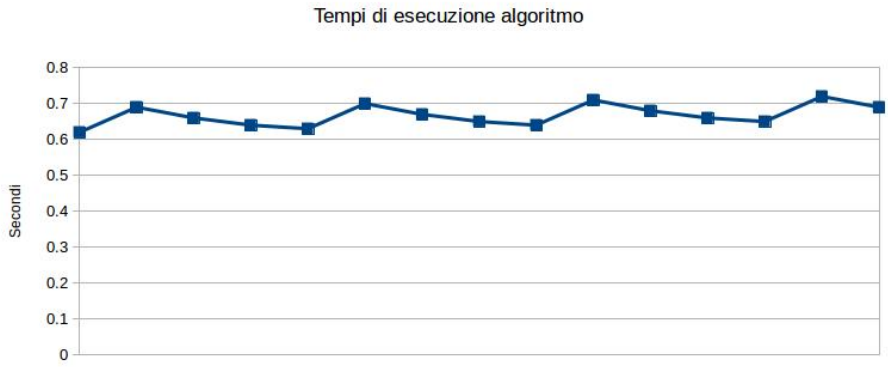

Figura 2.5: Grafico relativo ai tempi di esecuzione su BeagleBone

che la funzione che impiega il maggior tempo di esecuzione è l'individuazione del cartello nell'immagine in particolare il filtro di Canny. Infatti esso occupa circa il 50 - 60 % del tempo impegato dall'algoritmo. La funzione di trasformazione prospettica ha un tempo trascurabile ed infine la funzione di riconoscimento segnale non ha un tempo di esecuzione trascurabile ma comunque ridotto rispetto al filtro di Canny. Da notare che comunque il tempo di esecuzione è relativo per ogni esecuzione dell'algoritmo perchè dipende da vari fattori come per esempio altri processi.

2.5 Considerazioni

Un problema importante sul quale bisogna lavorare è che l'algoritmo non riesce a riconoscere i cartelli se questi sono posti a piú di un metro di distanza. Questo implica che il robot debba lavorare a basse velocità perchè un segnale abbia efficacia e che sia riconosciuto prima di averlo sorpassato. Un primo lavoro sta nel cercare di aumentare la distanza a cui possano essere riconosciuti i cartelli.

Un altro dato importante rilevato nel analisi dei dati è la velocita di esecuzione dell' algoritmo. Esso impiega circa 700 millisecondi per riconoscere un segnale. Se per esempio il robot si sposta alla velocità di 10 km/h esso riconosce il cartello solo dopo 0.7 secondi cioè quando il robot ha percorso 2 metri che comparato alle distanze che compie il robot nell'ordine dei 10 metri questo risulta essere eccessivo. Quindi si cerca di migliorare, cioè di ridurre il piú possibile il tempo di esecuzione.

Nel capitolo successivo verranno proposti e analizzati dei metodi per aumentare la velocità di esecuzione dell' algoritmo e aumentare la precisione

del riconoscimento di un cartello da una distanza superiore al metro. Inoltre si implementa anche un metodo per calcolare la distanza dal cartello.

Capitolo 3

Sviluppo e ottimizzazione algoritmo

In questo capitolo si espongono degli algoritmi per migliorare prestazioni e robustezza dell'algoritmo. Inoltre si propone un metodo per calcolare la distanza dal cartello.

3.1 Cambio cartelli

In questa sezione si analizza la problematica della distanza e si propone un metodo per risolverla.

3.1.1 Analisi problematica

Come analizzato nel precedente capitolo uno dei problemi principali è l'individuare il cartello in lontananza e riconoscere che tipo di cartello si tratta. Essendo la risoluzione non troppo elevata (640x480) quando ci si allontana dalla fotocamera le figure sono sempre piú piccole. Questo ovviamente ha delle ripercussioni sull' algoritmo. Di seguito verranno proposti dei metodi per cercare di risolvere questa problematica.

3.1.2 Cambio Cartelli

Ora passiamo ad analizzare il problema del riconoscimento delle figura all' interno del cartello.

I cartelli in lontananza hanno una figura interna facilmente confondibile o addirittura non si riescono a riconoscere. Questo è dovuto al fatto che la

risoluzione è bassa e quindi l'interno di un cartello presenta pochi pixel come si può notare nelle figure 3.1.

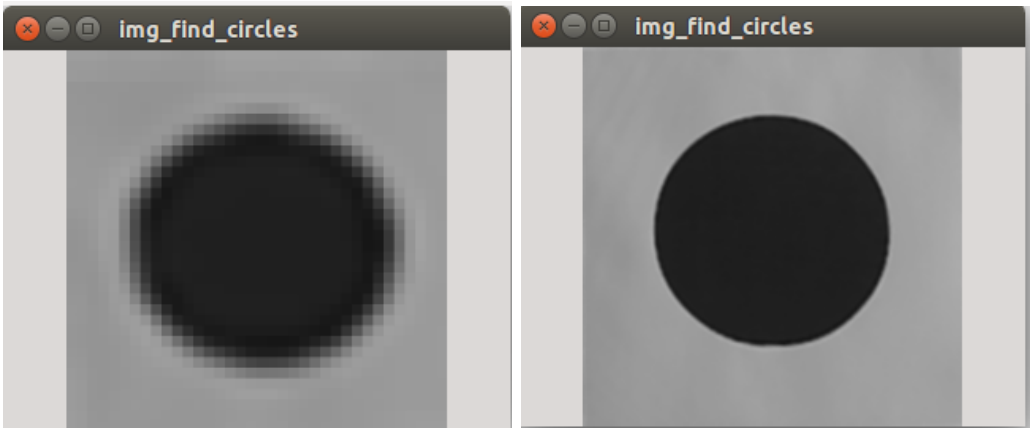

Figura 3.1: A sinistra l'interno di un cartello non riconosciuto, a destra invece uno riconosciuto

Queste immagini sono l'interno di un cartello a distanza di 3 metri. Come si può vedere quella a sinistra è poco nitida o comunque poco definita e non viene riconosiuta. Questo è dovuto al fatto che alla traformata di hough per riconoscere i cerchi serve una immagine nitida.

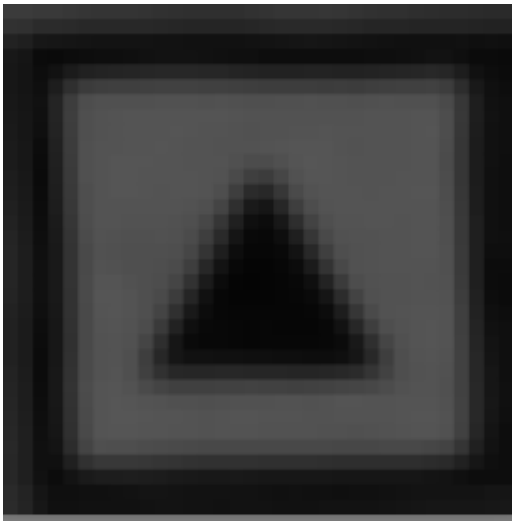

Figura 3.2: Triangolo non riconosciuto a causa della risoluzione

Come si puó vedere le due figure sono poco nitide e l'algoritmo ha un elevato tasso di errore nel riconoscerle perchè per riconoscere una figura bisogna avere una risoluzione piuttosto alta. Nell' immagine 3.2 addirittura il

cartello viene riconosciuto come cerchio. Per ovviare a questo inconveniente è stata cambiata la grafica dei cartelli.

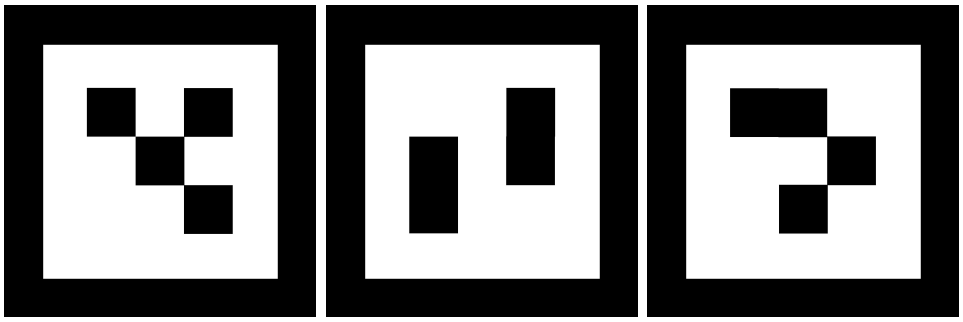

Figura 3.3: Nuovi cartelli

3.1.3 Analisi nuovi cartelli

Nella grafica dei nuovi cartelli è stato mantenuto il bordo nero e il fondo bianco che permette di trovarli facilmente nell'immagine. è stato cambiata invece la grafica interna come si puó vedere nella figura 3.3. In centro alla parte bianca del cartello c'è una griglia di 3x3 quadrati di uguali dimensioni dei quali solo alcuni sono neri. Il metodo utilizzato per decidere se un quadrato è nero o bianco è che si cerca di massimizzare la differenza fra i vari cartelli. Nel nostro caso i tre cartelli hanno una differenza tra uno e l'altro del 66%. Ciò significa che se si prendono due cartelli, solo 3 quadrati su 9 della griglia sono colorati nella stessa maniera.

è stato scelto questo tipo di cartello perchè ogni segnale differsice molto da un altro e quindi è difficile confondere i due anche se ci sono errori dovuti alla scarsa risoluzione.

3.1.4 Implementazione nuovo riconoscimento

L' algoritmo per l'implementazione è totalmente diverso da quello precedente. Mentre quello preesistente si avvaleva del filtri di Canny e cercava le forme geometriche dentro il cartello questo utilizza un piú semplice pattern matching ottenendo sia risultati migliori sotto forma di prestazioni che di percentuale di cartelli riconosciuti.

La funzione prende in input la porzione di immagine interna al cartello (ROI) in scala di grigi e il vettore contenente i cartelli conosciuti. Per prima cosa la porzione all'interno dell' immagine viene convertita in bianco e nero

Function `calcDifference`(*ROI,cartello*)

$originalWidth \leftarrow ROI.width$;
$originalHeight \leftarrow ROI.height$;
$ROI \leftarrow biancoNero(ROI)$;
$ROI \leftarrow resize(ROI, N)$;
$ROI \leftarrow cutBorderBalck(ROI)$;
$ROI \leftarrow cutBorderWhite(ROI)$;
if $ROI.width <= originalWidth || ROI.heigth <= originalHeight$ **then**
  return -1;
end
$ROI \leftarrow resize(ROI, N)$;
$ROI \leftarrow resize(cartello, N)$;
$uguali = contaPixelUguali(ROI, cartello)$;
return $uguali/pow(N, 2)$;

Algorithm 1: Pseudo codice dell'algoritmo

utilizzando la funzione di openCV cv::threshold(). Il livello di threshold è deciso facendo la media di livello di grigio di tutta l'immagine. Il vantaggio di avere un livello di threshold variabile e non fisso è che se l'immagine viene presa in un ambiente scuro esso si alza automaticamente discriminando il bianco dal nero.

A questo punto l'immagine viene ingrandita a una dimensione nota e si tagliano i bordi neri che potrebbero esserci come rimanenze del bordo del cartello e si taglia anche la parte bianca attorno alla griglia di modo che rimanga solamente la figura interna. Per tagliare i bordi neri si prendono in considerazione i pixel esterni che formano la cornice. Se nella cornice è presente almeno un pixel nero allora la si elimina e si prende in considerazione la nuova cornice eseguendo il passo precedente fintantochè la cornice è formata da soli punti bianchi. A questo punto si taglia la parte bianca attorno alla figura. Per eliminarla si esegue un procedimento analogo solo che in questo caso si taglia il bordo solo se la cornice ha i pixel colorati tutti di bianco. Il taglio dei bordi consente di estrarre la figura interna al cartello ma serve anche come filtro a falsi cartelli riconosciuti come tali dai passi precedenti dell'algoritmo. Un cartello non è tale con molta probabilità verrà tagliato tutto restituendo solo un pixel quando si taglia il nero perchè a meno che non abbia bordi bianchi esso continuerà a trovare del nero sul bordo fino a superare la grandezza minima dell'interno del cartello impostata al 70% della larghezza dell'algoritmo.

Adesso si portano sia i cartelli che l'immagine interna al cartello a una

grandezza uguale. A questo punto si ha solo la griglia interna al cartello e si passa all'effettivo riconoscimento di esso. Per identificare la griglia interna si esegue un semplice pattern matching fra cartelli e parte interna che cartello che vogliamo riconoscere. Per fare pattern matching basta fare uno xor pixel a pixel tra le due immagini da confrontare. Se un cartello ha una percentuale di pixel corrispondenti piú alta degli altri e piú alta di una soglia impostata del 70% verrà riconosciuto come cartello corrispondente.

3.2 Vanishing Point

3.2.1 Analisi ambiente

Analizzando i dati del veccio algoritmo si è notato che la funzione che tiene occupato il processore per più tempo è il filtro di Canny. La funzione di Canny è fortemente dipendente dalla grandezza dell'immagine. Una possibile soluzione è ridurre l'area di ricerca dell'immagine per incrementare le prestazioni della funzione e quindi ridurre i tempi di esecuzione.

I cartelli vengono sempre posizionati alla fine di un rettilineo e prima di una curva e mai durante o appena dopo di essa. Così facendo le immagini che dovranno essere analizzate saranno piú nitide per il semplice fatto che la telecamera sta facendo meno movimento laterale e le immagini risultano meno 'mosse' di conseguenza meno disturbate.

Analizzando i frame catturati dalla fotocamera frontale si puó notare che la posizione dei cartelli all' interno dell' immagine è abbastanza fissa. Infatti in quasi tutte le immagini i cartelli si trovano sulla linea dell' orizzonte dato che essi sono fissi su di esso. Piú ci si allontana e piú' i cartelli sono fissi sull orizzonte.

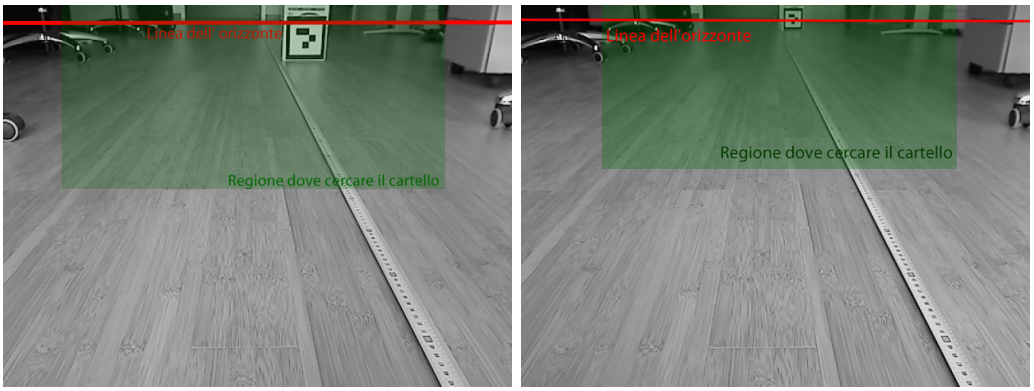

Figura 3.4: In rosso la linea dell'orizzonte e in verde la regione dell'immagine dove dovrebbe essere cercato il cartello

3.2.2 Punto sull' orizzonte

Nella sezione precedente è stato detto che il cartello è conveniente cercarlo prima sulla linea dell'orizzonte. Ma come fare riconoscere l' orizzonte? La soluzione è l' implementazione di un algoritmo che calcola il punto sull'orizzonte detto anche Vanishing Point [2].

Il vanishing point è un punto nell'immagine che è il risultato dell' intersezione del prolungamento delle linee parallele sul piano di immagine. Per esempio le linee bianche che stanno ai bordi di una strada sono parallele ma nell'immagine se si prolungano si incontreranno sulla linea dell' orizzonte. Sfruttando questo principio è possibile calcolare il Vanishing Point.

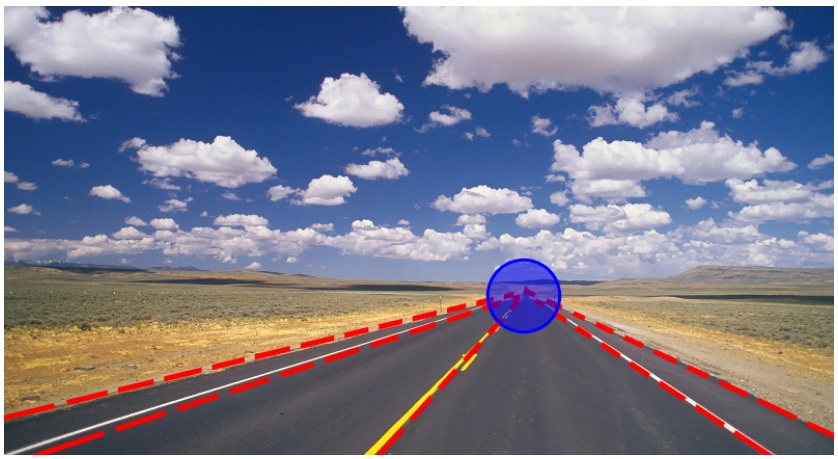

Figura 3.5: In rosso le linee che dovrebbero essere rilevate e in blu l'area dove approssimativamente esse si incontrano

3.2.3 Calcolo del Vanishing Point

Come detto in precedenza per calcolare il Vanishing Point servono le linee ai bordi della strada quindi per prima cosa bisogna estrarre i segmenti dall' immagine. Per fare ció è stata usata la funzione lsd() (line segment detection) che implementa un algoritmo di rilevazione dei segmenti descritti nel paper [3] e che prende in input l'immagine e restituisce un array di **double**. Gli elementi dell' array vanno presi 7 alla volta e vanno utilizzati come segue:

1. **x** del punto di partenza del segmento nell' immagine
2. **y** del punto di partenza del segmento
3. **x** del punto di arrivo del segmento

4. **y** del punto di arrivo del segmento
5. **w** lunghezza del segmento
6. **p** p è la precisione con cui è calcolato il segmento
7. **logNfa** $-\log_{10}(NFA)$

Successivamente per ogni segmento si calcola il punto medio di esso e la pendenza. I segmenti successivamente vengono filtrati. I segmenti devono:

- Essere piú lunghi di una certa lunghezza, impostata al 10% della larghezza dell'immagine, altrimenti potrebbero essere solo rumore o oggetti al bordo della strada.
- Avere una pendenza tra 10° e 70° questo perchè le linee orizzontali non vengono utilizzate e le linee verticali, cioè piu di 70°, potrebbero essere segmenti non appartenenti alla strada per esempio un edificio.

Figura 3.6: In nero si vedono i segmenti rilevati e filtrati

Adesso si deve determinare l'intersezione tra tutti i segmenti. Per calcolare l'intersezione tra due segmenti si calcola per ogni segmento la retta passante per esso e poi si calcola l'intersezione tra esse. Si scartano i punto di intersezione che escono dall'immagine. Per ogni intersezione si calcola anche un peso. Il peso è dato dalle due lunghezze dei segmenti elevate al quadrato

$$weight = (seg1.width * seg2.width)^2$$

Questo per assegnare un importanza maggiore all' intersezione di due segmenti piú lunghi. Il Vanishing Point temporaneo è calcolato come la media pesata dei punti di intersezione calcolati in precedenza.

$$\sum_{i \in intersezioni} i * \frac{i.weigth}{sommaWeigth}$$

Infine per calcolare il Vanishing Point effettivo si fa la media con il Vanishing Point calcolato nell'immagine precedente.

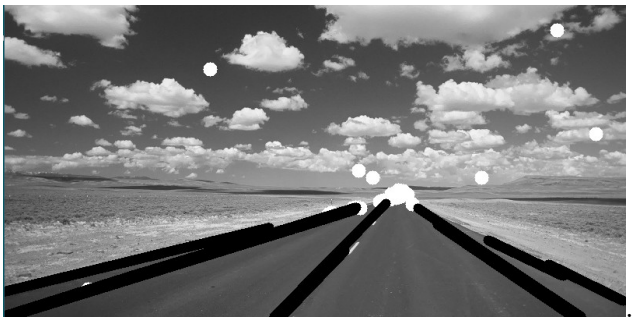

Figura 3.7: In bianco le intersezioni tra segmenti calcolate

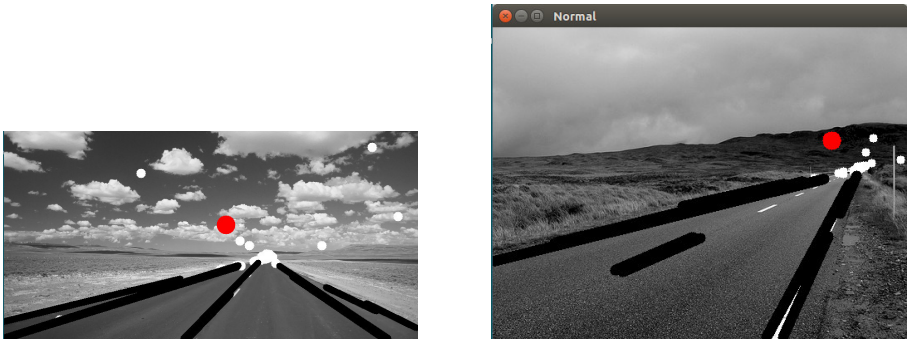

Figura 3.8: In rosso i Vanishing Point calcolati

3.2.4 Implementazione

L'algoritmo per il calcolo del punto sull'orizzonte viene chiamato prima di fare qualsiasi operazione sull' immagine. L'algoritmo, una volta calcolato il punto sull'orizzonte, controlla che essso sia nell'immagine perchè il punto sull'orizzonte potrebbe essere anche esterno ad essa. Se questo è esterno non si prende in considerazione, altrimenti si taglia l'immagine nel suo intorno. Il taglio viene eseguito tenendo una percentuale di pixel rispetto alla larghezza dell' immagine. Questa percentuale è fissata al 60% rispetto alla larghezza e anche all'altezza. Siccome il punto sull'orizzonte è quasi sempre calcolato nella parte alta come nella figura 3.2 la parte tagliata in altezza sarà quasi sempre minore al 60% riducendo cosí la superfice da analizzare successivamente aumentando le prestazioni.

Il calcolo del vanishing Point come vedremo nella prossima sezione ha un costo computazionale parecchio elevato di circa un secondo quindi è stato scelto di calcolarlo solo sulla prima immagine quando si arriva in rettilineo. Successivamente se si trova il cartello nell'intorno del punto sull'orizzonte allora si passa alla prossima immagine e su di esso non si ricalcola tenen-

do quello precedente, altrimenti si cerca il cartello in tutta l'immagine e si cercherà di ricalcolare il vanishing point nel prossimo frame.

Figura 3.9: A sinistra il frame originale con il vanishing point in bianco. A destra l'immagine nell'intorno del Vanishing Point

3.3 Distanza dal cartello

Dopo aver riconosciuto il cartello si passa alla parte piú importante dell'algoritmo: il calcolo della distanza dal segnale.

3.3.1 Analisi ambiente

I passi precedenti dell' algoritmo individuano il cartello restituendo i punti del rettangolo che lo contengono. Per calcolare la distanza sapendo la grandezza del cartello a priori, si potrebbe ricavare la larghezza in pixel quindi in funzione di essa si puó calcolare la distanza. Il problema, comunque, rimane la bassa risoluzione della telecamera. Infatti dopo aver preso delle immagini sperimentali di cartelli posti a 2 e 3 metri si sono analizzate le larghezze in pixel dei due cartelli e si è notato che essi differivano di un pixel o addirittura in certi casi erano identiche. Un diverso approccio sfrutta IPM (Inverse Perspective Mapping).

3.3.2 L' algoritmo

La vista prospettica dell'immagine catturata distorce la forma reale della strada. L' Inverse Prospective Mapping è un processo che serve appunto per eliminare eventuali distorsioni, rimappando l'immagine attraverso una rototraslazione. Dopo l' IPM essa appare come se fosse stata presa da una

telecamera perpendicolare al terreno. Questo algoritmo di rettifica è già stato utilizzato durante questo progetto [4] per riuscire a trovare la linea nera sul terreno e riuscire così a pianificare il percorso.

A noi peró per calcolare la distanza dal cartello non serve rimappare tutta l'immagine ma solo un punto cioè il punto di strada dove il cartello è perpendicolare ad esso. Per fare ció prima di tutto abbiamo bisogno di ricavare tale punto. Siccome non c'è modo di rilevare, oltre al cartello, anche l'altezza del suo supporto è stato deciso di appoggiare i cartelli a terra e di non tenerli sollevati. Così facendo trovare il punto sul terreno risulta molto piú facile, basta infatti prendere il centro della base del cartello, trovando così dove si appoggia sul terreno e ottenendo il punto da noi desiderato. Ora basta

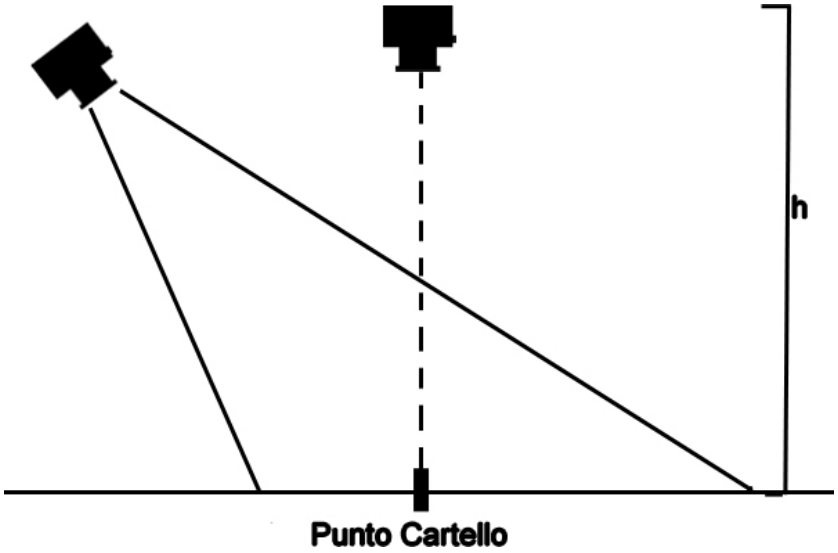

Figura 3.10: A sinistra la camera telecamera reale con il suo campo di visione e al centro la telecamera virtuale perpendicolare al punto sul terreno dove appoggia il cartello.

passare il punto all' algoritmo di rettifica. La rettifica crea una telecamera virtuale perpendicolare al terreno ad una altezza nota. Tenendo conto dell' angolo della camera reale rispetto al terreno la rettifica ci restituisce la x e la y del punto nell'immagine come se fosse stata presa da quella telecamera.

Adesso abbiamo il nostro punto nell'immagine dall'alto e dobbiamo ricavare quello nella realtà. Per fare ció come si nota nella figura 3.11 basta fare una semplice proporzione sia per la coordinata x che per la coordinata y

$$xi : focal = x : h$$
$$yi : focal = y : h$$

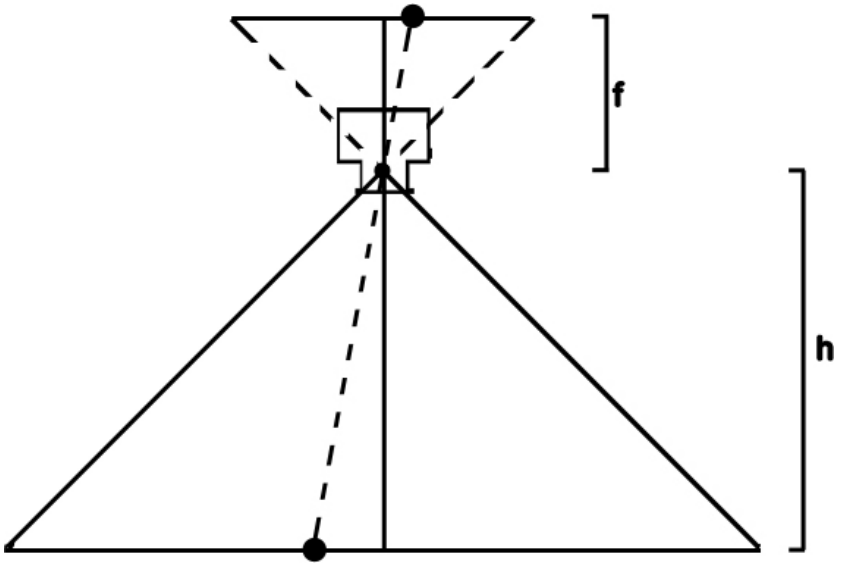

Figura 3.11: f distanza focale dell'immagine, h l'altezza dell'immagine dal terreno.

Dove xi e yi sono le coordinate del punto nell'immagine rettificata. Focal è la distanza focale dell'obbiettivo e h è l'altezza della telecamera virtuale. Invece x e y sono le coordinate del punto nella realtà da calcolare sul piano del terreno che sono anche le coordinate di posizione ripetto al robot. In pratica se si conosce la distanza del piano dell'oggetto, la posizione dell'oggetto nell'immagine e la distanza focale, allora si riesce a calcolare la distanza dell'oggetto. Infine bisogna sommare l'offset della telecamera rispetto al centro del robot e abbiamo la posizione del segnale rispetto alla posizione del robot. Se si vuole sapere la distanza effettiva basta applicare il teorema di pittagora per ricavare la diagonale. Di seguito lo pseudo codice.

Function `computeDistance(`$Point\ a$`)`
  $Point\ \ p \leftarrow rectifier.rettificaPunto(a, camera)$;
  $float\ \ focal = camera.getFocal()$;
  $Point\ \ x \leftarrow a.x * h/focal$;
  $Point\ \ y \leftarrow a.y * h/focal$;
  $x \leftarrow 0.1869 - y$;
  $y \leftarrow -0.0219 - x$;
  return $sqrt(pow(x, 2) + pow(y, 2))$;

Algorithm 2: Pseudo codice dell'algoritmo di calcolo distanza

3.3.3 Implementazione

Per implementare l'algoritmo per il calcolo della distanza è stata utilizzata la libreria Eigen[1] che offre funzioni per calcoli matematici necessari per l'algoritmo di rettifica delle immagini.

L'algoritmo calcola la distanza su ogni immagine dove è stato trovato il cartello e se la tiene in memoria. Se non è stato trovato, incrementa una variabile che serve a tener conto da quante immagini il cartello non è stato trovato. Questa ultima variabile serve per sapere da quanto non è stato trovato il cartello e quindi per sapere quanto è affidabile l'ultima distanza calcolata. Piú aumenta, piú il robot ha fatto strada e quindi sempre meno affidabile.

[1] `http://eigen.tuxfamily.org/`

Capitolo 4

Esperimenti

In questo capitolo sono presentati i dati raccolti relativi a prestazioni, affidabilitá e precisione dell' algoritmo.

4.1 Test di precisione riconoscimento sui nuovi cartelli

I cartelli utilizzati sono grandi 21 centimetri di lato come la larghezza del lato corto di un fogio A4 e sono posizionati perpendicolari al terreno e appoggiati su di esso. Per prima cosa sono stati prese delle immagini da 1 a 6 metri ogni

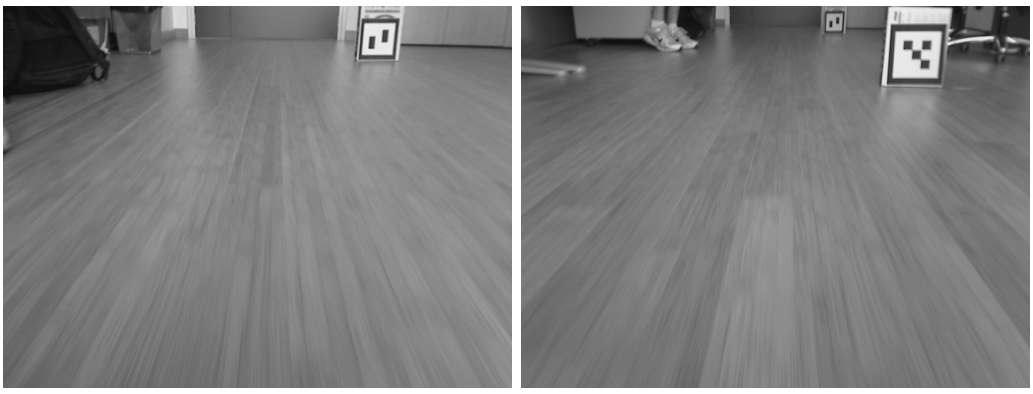

Figura 4.1: Esempi immagini prese in movimento

0.5 metri con il robot fermo. Successivamente sono state prese con il robot in movimento e con 2 cartelli posizionati a 3 e a 6 metri dal punto di partenza. Ogni prova é stata ripetuta piú volte cambiando i cartelli e nel caso del robot in movimento la prova è stata ripetuta a 3 diverse velocità.

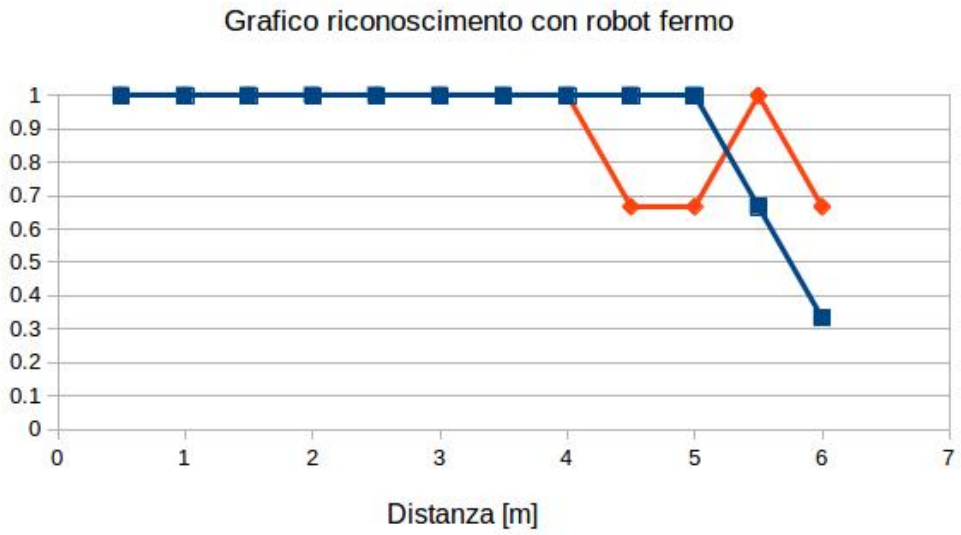

Figura 4.2: In blu la percentuale di riconoscimento della ROI. In rosso la percentuale di riconoscimento del cartello quando la ROI è già stata trovata.

Velocità	Riconosciuti giusti	Riconosciuti sbagliati	% Non trovati
0.5	94%	0 %	6 %
0.8	81%	0 %	19 %
1	80%	0%	20 %

Tabella 4.1: Percentuali di riconoscimento per velocità

I risultati sono presentati di seguito

Analizzando il grafico nella figura 4.1 si nota un sostanziale miglioramento di precisione nel riconoscimento. Infatti le immagini vengono riconosciute al 100% fino a quattro metri di distanza. Passati i quattro metri la precisione comincia a diminuire ma rimane accettabile non scendendo sotto il 60% di accuratezza fino ai 6 metri. Oltre i sei metri non sono stati effettuati ulteriori test.

Dai dati raccolti nella tabella 4.1 si nota che quando il robot va a velocità massima ha una precisione dell' 80% di segnali riconosciuti su tutti i frame presi; invece del 94% quando il robot va a metà della velocità. Questa differente precisione è dovuta alle diverse velocità del robot. Infatti analizzando le immagini prese ad entrambe le velocità si nota che i frame presi con la velocità massima risultano più sfuocati e di qualità minore; tuttavia questo non incide ampiamente sul risultato del test, infatti la differenza di risultati è minima. Guardando poi alla terza colonna della tabella si conclude che un

cartello non viene mai scambiato con un altro cioè se viene riconosciuta una ROI il suo interno non viene mai scambiato con quello di un altro cartello al massimo non lo identifica.

Figura 4.3: Esempi di cartelli riconosciuti con robot fermo.

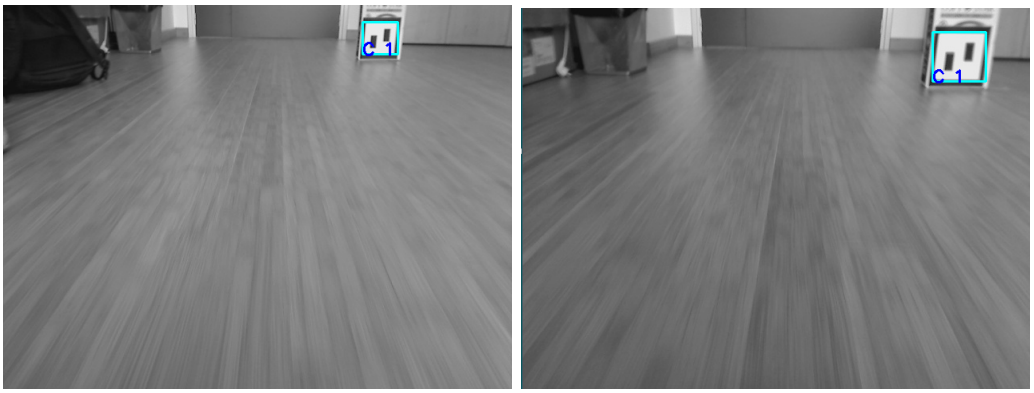

Figura 4.4: Esempi di cartelli riconosciuti con robot in movimento.

4.2 Test delle prestazioni

Adesso passiamo ad analizzare le prestazioni dell'algoritmo per calcolare il punto sull'orizzonte.

Per fare questi test sono state utilizzate delle immagini prese con il robot in movimento una di seguito all' altra partendo da una distanza dal cartello di 6 metri. La grandezza dei segnali é pari al lato corto di un foglio A4. Purtroppo le immagini sono state prese in laboratorio e non all'aperto su una strada peró il vanishing Point viene calcolato correttamente anche aiutato

dalle righe del pavimento che sono parallele alla direzione del robot. Sono state fatte 9 prove con il robot in movimento a diverse velocitá. Prima é stato fatto cercare il cartello usando l'algoritmo spiegato precedentemente sul vanishing Point e poi senza calcolarlo. Quello che é emerso é riportato nel grafico seguente.

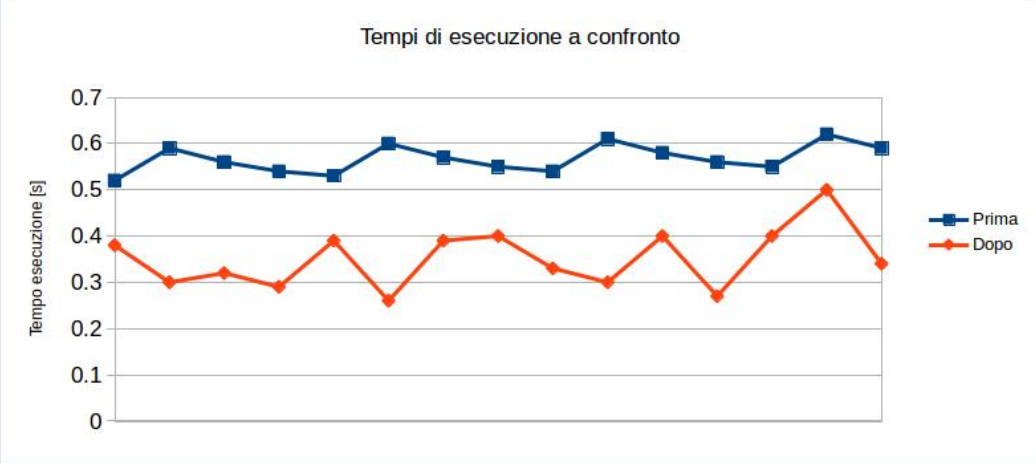

Figura 4.5: Grafico confronto tempi di esecuzione con e senza Vanishing Point.

Quello che ne risulta analizzando i dati è che si ha un guadagno del 38% quando si cerca un' immagine nell'area intorno al Vanishing Point. Purtroppo però la funzione che calcola il punto sull'orizzonte ha complessità computazionale molto elevata, soprattutto la funzione line segment detection che serve a calcolare i segmenti nell'immagine. Infatti il tempo impiegato dall' algoritmo del calcolo è di ~1 secondo. Come stato detto nel capitolo precedente è stato deciso dunque di far calcolare all' algoritmo il Vanishing Point solo sulla prima immagine e non su tutte. Esso viene ricalcolato solo nel caso in cui nell'intorno di esso non sia presente il cartello.

4.3 Test del calcolo distanza

Infine passiamo ad analizzare i risultati sul calcolo della distanza del cartello. Per questi test sono state utilizzate le immagini già usate in precedenza per gli altri test ma solo quelle prese con il robot fermo.

Dai dati raccolti emerge che l'algoritmo per alcune distanze ovvero quelle inferiori ai 3 metri risulta avere una buona precisione infatti l'errore massimo rilevato è dell'ordine di 10 centimetri. Tuttavia se si superano i 3 metri l'errore incrementa con conseguente perdita di precisione. Questa perdita di

Reali [m]	Calcolate [m]	Differenze [m]	Differenze al quadrato
1	0.93	0.07	0.0049
1.5	1.59	0.09	0.0081
2	1.99	0.01	0.0001
2.5	2.52	0.02	0.0004
3	3.07	0.07	0.0049
3.5	3.66	0.16	0.0256
4	4.22	0.22	0.0484
4.5	5.16	0.6	0.36
5	5.39	0.39	0.1521
5.5	6.42	0.92	0.8464
		Varianza	0.14509
		Deviazione standard	0.255

Tabella 4.2: Confronto delle distanze reali e calcolate dall' algoritmo

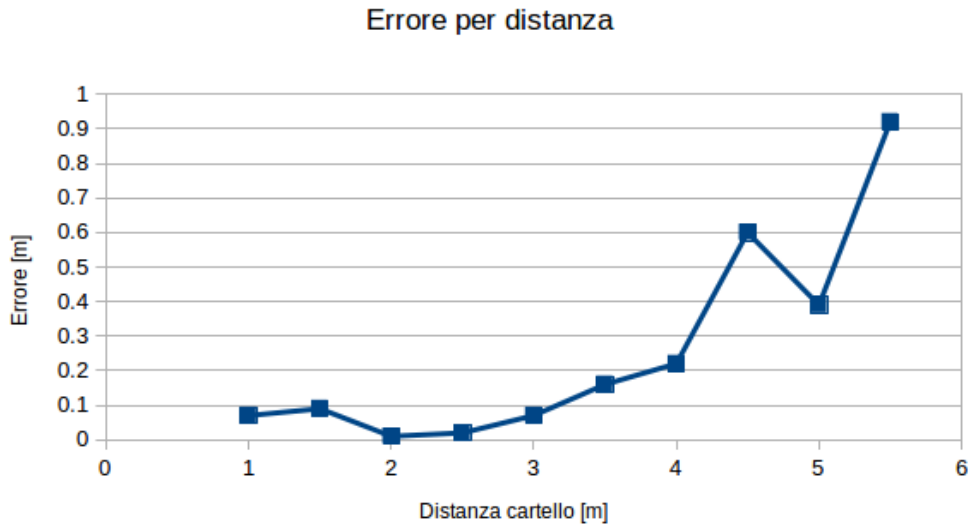

Figura 4.6: Grafico errore della distanza calcolata rispetto a quella reale

precisione è dovuta a due fattori principali: l'algoritmo di rettifica e la risoluzione della telecamera. L'algoritmo di rettifica introduce degli errori perchè più la distanza aumenta più si perde precisione nel rettificare le immagini. Questo è dovuto al fatto che quando si tenta di rettificare una parte dell'immagine vicina alla telecamera si hanno più punti a disposizione, invece più la distanza aumenta più si perde informazione nell'immagine e l'errore aumenta. Invece il problema relativo alla risoluzione è dovuto al fatto che incide

sia sull' algoritmo di rettifica che sul calcolo del punto sulla base del cartello. Quando la distanza del cartello dalla telecamera è più ampia, il cartello verrà rappresentato con meno pixel nell'immagine, invece se è vicino verrà rappresentato con molti più pixel. Ciò comporta che se l'errore nel calcolo del punto alla base del cartello è di una piccola percentuale di pixel rispetto alla totalita di pixel nell'immagine, l'errore sarà molto più ampio quando il cartello è distante. In conclusione si è deciso di tenere valida, per la correzione della posizione del robot nel percorso, la distanza calcolata solo se essa è inferiore ai 3 metri perchè garantisce un errore massimo di 10 centimetri.

Capitolo 5

Conclusioni

La tesi descrive dei metodi per migliorare vari aspetti riguardo il riconoscimento di cartelli ed espone un metodo per calcolarne la distanza da esso. Siamo partiti da dei requisiti che deve avere l' algoritmo di riconoscimento descritti nel primo capitolo e abbiamo analizzato l'algoritmo esistente e fatto dei test di affidabilità e velocità. Dopo aver analizzato i test si è deciso come sviluppare il progetto per arrivare ai requisiti fondamentali per una applicazione real time. É stato deciso di concentrasi sul migliorare l'affidabilità e prestazioni. Inoltre è stato deciso di implementare una nuova funzione che permette di calcolare la distanza dal cartello. Poi nel quarto capitolo sono stati presentati i dati raccolti dagli esperimenti eseguiti per testate i suddetti requisiti finali che l'algoritmo deve avere.

I risultato dei test è soddisfacente sotto il punto di vista della distanza di riconoscimento. Infatti i nuovi cartelli con il rispettivo nuovo algoritmo di riconoscimento vengono riconosciuti con discreto successo fino a 6 metri di distanza rispetto ai 2 metri del codice di partenza. Inoltre si è migliorato sotto l'aspetto di affidabilità cioè che un cartello non viene mai scambiato con un altro. Si sono ottenuti buoni risultati anche da parte della velocità di esecuzione. Se si cerca un cartello nell'intorno del punto sull'orizzonte si hanno dei miglioramenti in prestazioni di ~38% anche se il tempo per il calcolo è oneroso dato dal fatto che la libreria Line Segment Detection ha un elevato carico computazionale. Come ultima cosa i test per il calcolo della distanza hanno dato ottimi risultati se il cartello è posto a meno di 3 metri dalla robot invece oltre i 3 metri la precisione cala drasticamente quindi è stato deciso per il momento di tenere in considerazione i dati relativi a cartelli posti a meno di 3 metri.

5.1 Lavori futuri

Un aspetto importante che dovrebbe essere migliorato sono le prestazioni dell'algoritmo per il calcolo del Vanishing Point, in particolare la funzione per il calcolo dei segmenti nell'immagine la quale prende più tempo nell' algoritmo. Inoltre sarebbe opportuno testare la precisione di calcolo del Vanishing Point quando il robot sarà pronto per andare in strada. Un ulteriore step sarebbe quello di migliorare la precisione dell'algoritmo nel calcolo della distanza dal cartello posto a distanze superiori i 3 metri. In futuro con lo sviluppo di schede embedded sempre più potenti e il conseguente aumento di prestazioni, congiunto all'utilizzo di GPGPU (Genral purpose GPU) che servono a parallelizzare i processi, consentirebbe di rendere più robusto l'algoritmo senza influire sui tempi di calcolo. Una scheda più potente potrebbe, per esempio, consentire di aumentare la risoluzione della telecamera mantenendo i tempi di calcolo contenuti.

Ringraziamenti

Desidero ringraziare il Prof. Luigi Palopoli e Federico Moro che mi hanno seguito e consigliato durante lo sviluppo di questo progetto.

Un sentito ringraziamento va a mia mamma mio papà e mio fratello che mi hanno sempre aiutato, sostenuto e incoraggiato.

Ringrazio la mia ragazza Francesca per essermi sempre stata sempre a fianco anche nei momenti difficili.

Esprimo la mia gratitudine per i miei compagni di corso: Giulio, Alex, Roberto, Giovanni, Vincenzo e a tutta la mia compagnia per avermi accompagnato durante questi 3 anni di università.

Un ultimo ringraziamento va al mio ex coinquilino Christian per avermi sopportato per ben 3 anni di convivenza.

Bibliografia

[1] J. Canny. A computational approach to edge detection, ieee trans. on pattern analysis and machine intelligence, 8(6), pp. 679-698, 1986.

[2] Thomas Bucher Thorsten Suttorp. Robust vanishing point estimation for driver assistance, 2006.

[3] Jean-Michel Morel Rafael Grompone von Gioi, Jeremie Jakubowicz and Gregory Randall. Lsd: A fast line segment detector with a false detection control, 2010.

[4] Moro Federico. Vision-based robust path reconstruction for robot control, 2013.