ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

Corso di Laurea in Ingegneria Informatica

TESI DI LAUREA

in

Calcolatori Elettronici T

VALUTAZIONE SPERIMENTALE DI METODOLOGIE DI RETTIFICAZIONE E IMPATTO SU ALGORITMI DI VISIONE STEREO

CANDIDATO
Luca Buratti

RELATORE:
Prof. Stefano Mattoccia

CORRELATORI
Dott. Matteo Poggi
Dott. Paolo Di Febbo

Anno Accademico 2015/2016

Sessione II

SOMMARIO

Capitolo 1. Introduzione

1.1 Computer Vision e Visione Stereo

La computer vision è quel ramo dell'informatica che si occupa di acquisire ed elaborare informazioni 3D provenienti dall'ambiente circostante, al fine di poter raggiungere determinati scopi, dalla navigazione autonoma di robot all'identificazione di oggetti e tanto altro; in particolare la visione stereo [1][2][3], che ne è una sua tecnica, si occupa di simulare il comportamento dell'occhio umano mediante l'utilizzo di due o più telecamere.

Lo scopo di questa tecnica è dedurre informazioni "dal mondo circostante" e in particolar modo dedurre il senso di profondità degli oggetti della scena, un concetto che però deve essere estratto da immagini che sono ovviamente bidimensionali.

E' necessario quindi passare dal mondo 2D delle immagini a quello 3D della scena reale.

Per inferire immagini di questo tipo si fa uso di un sistema stereo, ovvero un sistema composto da almeno due telecamere.

Il perché non si possa usare una sola camera è abbastanza evidente: due punti che si trovano sulla stessa linea, proiettati nel piano immagine della telecamera, coincidono, andando a perdere il loro contenuto informativo sulla differenza di profondità; ciò non accade se si va ad usare due telecamere, come si evince dalla immagine 1.1.

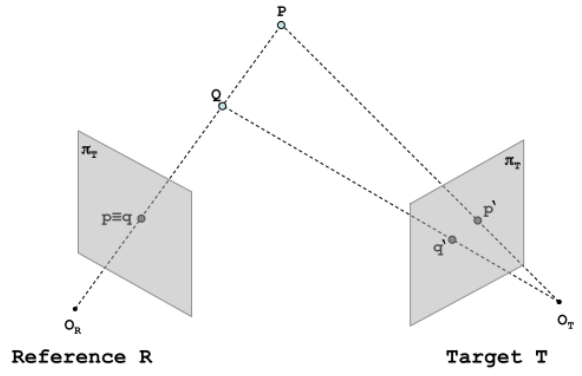


Figura 1.1 Schematizzazione di un sistema stereo. Usare due telecamere è molto comodo per fare uso in seguito della triangolazione

La visione stereo deduce il senso di profondità di un punto in base alla posizione che questo va a occupare nelle immagini (che in questa tesi saranno definite *left e right)* catturate dalle telecamere.

Per poter ottenere dunque una corretta idea della scena 3D, come si può immaginare sempre dalla figura 1.1, è necessario risolvere quello che viene definito *correspondence problem*, ovvero l'identificazione di un punto appartenente all'immagine *left*, cioè quella ottenuta dalla telecamera di sinistra, e del corrispondente nell'immagine *right*, cioè quella ottenuto dalla telecamera di destra.

In parole più semplici, riferendosi alla figura 1.2 bisogna fare in modo che la proiezione del punto Q sull'immagine *left (o reference)* sia localizzato all'interno dell'immagine *right (o target)* in modo da poterne triangolare la posizione, ottenendo quindi la distanza.

L'identificazione di un punto tra le due immagini è facilitata dal supporto che è fornito dalla geometria epipolare: prima si era infatti detto che la ricerca la si spostava dal mondo reale, 3D, a quello delle immagini in 2D; grazie a questa tecnica riusciamo a spostare il problema di corrispondenza tra immagine *left* e *right* addirittura al mondo 1D.

Il vincolo epipolare infatti afferma che se abbiamo due punti P e Q che giacciono sulla stessa retta e dunque le loro proiezioni coincidono nel piano immagine della telecamera sinistra allora si è certi che la soluzione del *correspondence problem* per tutti i punti che stanno sulla sopracitata retta avviene su una sola linea del piano immagine destro.

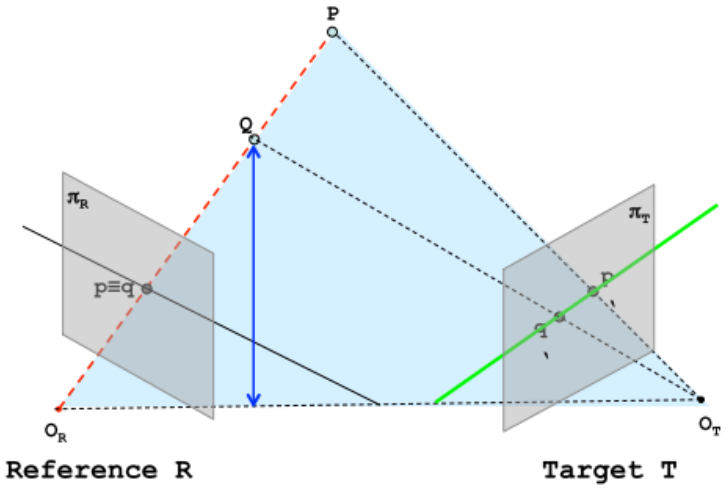


Figura 1.2 Proiezione di punti sui piani immagine delle telecamere del sistema stereo utilizzato

Un sistema così configurato rende però la ricerca non particolarmente comoda, dato che avviene su linee oblique: sicuramente sarebbe meglio avere linee epipolari dritte e parallele tra loro.

Per raggiungere tale obiettivo si fa uso di quello che è detto sistema stereo in forma standard, ovvero le telecamere sono ora allineate e caratterizzate dalla stessa distanza focale. Tale sistema è realizzato come mostrato in figura 1.3

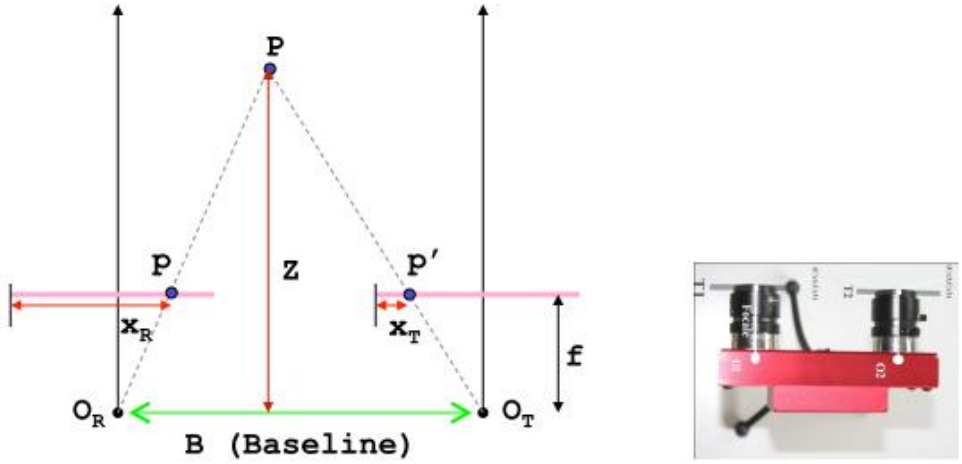


Figura 1.3 Sistema stereo in forma standard: calcolo della profondità del punto P

Si noti che è geometricamente dimostrabile che la distanza di un punto della scena dalla telecamera è inversamente proporzionale alla distanza orizzontale delle proiezioni di tale punto sui piani immagine (dimostrato da Xr e Xt in figura 1.3)

Si introduce ora il concetto di disparità: un valore che è inversamente proporzionale alla distanza dei punti della scena dalle telecamere.

La disparità dunque è calcolata come la differenza tra Xr e Xt. Il calcolo di questo valore, per ogni *pixel* dell'immagine, produce la mappa di disparità, una immagine solitamente in scala di grigi che codifica il contenuto informativo di quanto calcolato utilizzando l'intensità dei pixel: i punti lontani saranno associati a valori di intensità minore, ovvero più scuri, quelli più vicini saranno associati a pixel di intensità maggiore, dunque più chiari.

Una verifica sull'effettivo valore della differenza tra le coordinate x può essere fatta variando il punto P della figura 1.3: allontanandolo dai piani immagine si avrà che Xr diminuisce e Xt aumenta, andando a portare la differenza a valori più bassi,

informazione codificata con pixel di intensità minore; l'effetto contrario lo si ottiene avvicinando il punto P.

Conoscere la disparità, insieme alla conoscenza di parametri dipendenti dal sistema e che sono rappresentati in figura 1.3 (distanza tra i due centri ottici Or e Ot e distanza focale) consente, mediante operazioni trigonometriche, di ottenere Z, cioè la profondità: $Z = \frac{B*f}{Xr - Xt} = B * \frac{f}{d}$ con d che indica appunto la disparità. Il calcolo di Z per ogni punto genera la *depth map*, una immagine che in base alle tonalità dei colori rende l'idea della distanza degli oggetti nella immagine acquisita.

Come si è detto, per avere un calcolo della disparità corretto è necessario un sistema in forma standard in modo che la ricerca dei pixel per il *matching stereo* avvenga su una stessa linea dritta; purtroppo la presenza di errori nel configurare un sistema stereo è naturale (disallineamento tra lenti, distorsione prodotto dalle lenti stesse,…) e quindi ci si deve aspettare, almeno inizialmente, linee epipolari oblique: si capisce qui la necessità di avere una specie di filtro [4][8] che intervenga prima di iniziare a risolvere il problema della corrispondenza.

Tale filtro si occupa di correggere virtualmente problemi che per forza di cose sorgono nel mondo reale, infatti ad esempio il disallineamento tra le telecamere non può mai essere veramente nullo.

Questa specie di filtro astratto è realizzato dal processo di rettificazione e prende in input l'immagine con le linee oblique e fornisce in output una immagine con tali linee allineate con le righe dell'immagine, cioè come se la scena fosse stata catturata da un sistema perfettamente allineato, e si fa carico anche dell'eliminazione delle distorsioni delle lenti; si comprende quindi come la visione stereo sia attuata grazie ad una serie di operazioni eseguite in successione: un tipico schema a stadi, per cui si parla di *pipeline* di visione stereo.

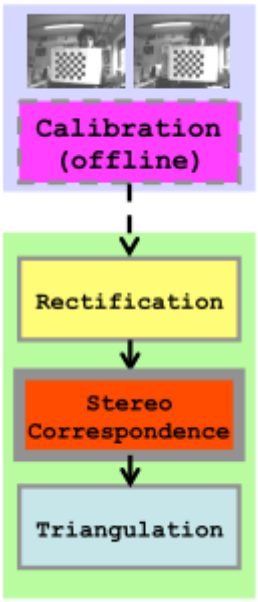


Figura 1.4 Stadi di una pipeline di visione stereo

1.2 Obiettivi

Questa tesi si occupa della parte della *pipeline* mostrata in figura 1.4, a cavallo tra gli stadi di rettificazione (su cui si trova una spiegazione al capitolo 3) e *stereo correspondence*; in particolare l'obiettivo è di identificare una configurazione di algoritmi di rettificazione che abbiano un buon rapporto tra qualità della rettificazione e impiego delle risorse utilizzate. Tale tesi infatti è inserita in un contesto più ampio, volto all'identificazione di algoritmi e parametri che risultino idonei ad una implementazione su FPGA, dove l'uso delle risorse (memoria, componenti hardware,…) va ben calcolato.

La sperimentazione è stata svolta sul *dataset* KITTI 2015 [5], un insieme di 200 immagini *left e right*, perfettamente rettificate. Tale *dataset* è stato realizzato montando telecamere su un'automobile che girava per le strade di una città: la sperimentazione è svolta quindi su immagini "del mondo reale" e non create ad arte in laboratorio dunque i risultati potrebbero essere quindi meno buoni di quelli ottenuti negli studi sul dataset *Middlebury* [9], che è il punto di partenza di questa tesi.

Sulle immagini KITTI sono state già svolte operazioni di *matching stereo* [7] mediante diversi algoritmi e ne sono state tratte conclusioni su quali siano quelli che minimizzino l'errore medio (il cui significato è introdotto più avanti) nella procedura di *stereo*

correspondence. Tale errore medio è l'obiettivo a cui si tende quando le immagini su cui si fa il *matching stereo* non siano più prive di distorsioni, ma siano state rettificate con algoritmi effettivamente implementabili su FPGA, cioè opportunamente modificati, anche a costo di perdere in qualità, al fine di utilizzare meno risorse.
In questa tesi quando si parlerà di questi algoritmi lo si farà riferendosi ad algoritmi di rettificazione "semplificata".
E' sottointeso dunque che vi siano algoritmi di rettificazione che risolvano i problemi di cui si è detto in modo quasi perfetto, ad esempio quelli resi disponibili dalla libreria OpenCV: il loro problema è l'implementabilità su dispositivi *embedded*, visto che richiedono largo uso di risorse.
L'obiettivo finale quindi è costruire un *set* di risultati che, per ogni algoritmo di rettificazione "semplificata" utilizzato, confronti l'errore medio ottenuto dagli algoritmi di *matching stereo* con l'errore medio "ideale" definito in precedenza.

Capitolo 2. Strumenti, linguaggi e ambiente di sviluppo

Gli strumenti utilizzati nello svolgimento di questa tesi sono stati i seguenti:

- OpenCV [6], una libreria open-source creata per il mondo della computer vision e dell'elaborazione di immagini. Caratterizzata dall'essere multipiattaforma, è stato utilizzato infatti per questa tesi sia in ambiente Linux che in ambiente Windows. Di questa libreria sono state utilizzate le funzioni di salvataggio e apertura immagini, di analisi dei pixel di immagini e anche funzioni di più alto livello che presa un' immagine offrano servizi per svolgere i calcoli necessari alla realizzazione di calibrazione, rettificazione e *matching stereo*
- *Stereo Software* [7], un progetto realizzato in C/C++, che fornisce un'interfaccia con cui si possono lanciare i vari algoritmi di *matching stereo.* Tale progetto è stato utilizzato in ambiente Linux e modificato al fine di lavorare sulle immagini rettificate tramite algoritmi idonei ad una implementazione su FPGA
- *FPGA map computation analysis* [8], un progetto realizzato in Python come tesi di laurea magistrale da Paolo di Febbo, che fornisce i vari algoritmi di rettificazione e vari parametri da utilizzare su questi per capire fino a quanto ci si può spingere nel risparmiare risorse. Tale progetto fornisce anche l'utile funzione di "distorsione artificiale" delle immagini, realizzata mediante una serie di funzioni Python/OpenCV
- KITTI 2015, un *dataset* composto da 200 foto (*left e right)* perfettamente rettificate, su cui si è svolta l'elaborazione di questa tesi. Tale *dataset* fornisce anche immagine di *ground-truth* per ogni immagine
- *RectificationLibrary* [4], un progetto sviluppato come tesi di laurea da Vincenzo Villani. Tale progetto, sviluppato per Visual Studio al fine di realizzare una libreria di funzioni utili alla rettificazione di immagini, offre una importante analisi sull'algoritmo di rettificazione semplificata *sampling*, di cui si discuterà più avanti in questa tesi. Su questo progetto è stata fatta la prima fase di studio riguardante la rettificazione.

Capitolo 3. Rettificazione e Calibrazione

3.1 Tipologia di distorsioni e correzione

Le lenti delle telecamere di cui si fa utilizzo per l'acquisizione delle immagini sono responsabili di due diversi tipi di distorsione:

- distorsione tangenziale, ovvero le lenti sono disallineate tra di loro e quindi si ottengono, come era stato spiegato precedentemente nel capitolo "Introduzione", delle linee epipolari oblique, che complicano il lavoro di *stereo correspondence*
- distorsione radiale, ovvero le lenti, che per natura non hanno forma regolare, tendono a curvare le linee della scena che in realtà sono dritte. Tipicamente tale distorsione è più evidente allontanandosi dal centro immagine. Tale distorsione però non è così presente per lenti normali, se non nei bordi, e si nota molto meglio se si fa uso di lenti particolari come le *wide angle.*

La distorsione introdotto dalla lente è descritta da una formula matematica che può essere invertita al fine di correggere il problema. Grazie a questa formula è possibile associare un punto dell'immagine distorta a un punto dell'immagine corretta e ricalcolarne le coordinate giuste: tuttavia questa operazione è impraticabile in hardware per via delle risorse richieste e dunque lascia spazio alla tecnica chiamata *inverse mapping*, che a partire dai punti dell'immagine corretta trova i punti corrispondenti nella distorta. Fino a qui però il costo dell'operazione è immutato: la miglioria è il fatto che ora si può esprimere ogni punto dell'immagine distorta come scostamento (più comunemente detto *displacement)* dal punto "corretto". Tale scostamento è espresso in coordinata x e y e visto che è fornito per ogni punto dell'immagine viene naturale organizzare sotto forma di tabella (detta anche matrice) tutti gli scostamenti di tutti i punti.

Le tabelle di *displacement* sono ottenute mediante calibrazione della telecamera. Tale procedura è una operazione da fare seguendo determinati passi, che in questa tesi non saranno enunciati in maniera approfondita, per impostare al meglio il sistema stereo; OpenCV fornisce funzioni per la generazione di queste tabelle (una per la coordinata x, una per la coordinata y).

Il processo di correzione delle distorsioni, ricapitolando, si può schematizzare come rimozione distorsioni generate dalle lenti e allineamento delle rette su cui si trovano i

punti rispettivamente dell'immagine *left* e dell'immagine *right* (parte detta di rettificazione*)*

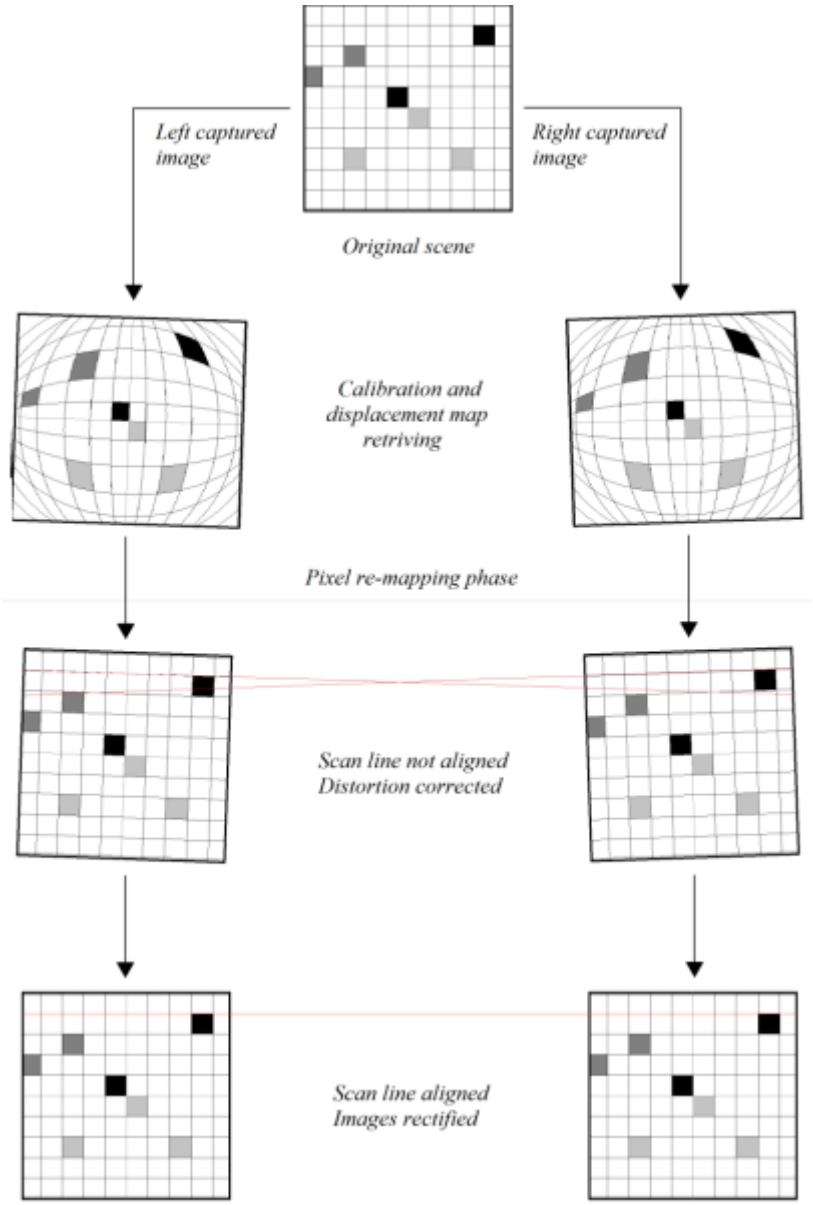


Figura 3.1 Processo di rettificazione: eliminazione delle distorsioni e allineamento delle linee epipolari

3.2 Dall'*inverse mapping* alla rettificazione: utilizzo dell'interpolazione bilineare

L'utilizzo delle tabelle del *displacements* non è sufficiente per correggere un'immagine distorta: infatti non tutti i pixel della immagine corretta puntano correttamente ad un

pixel dell'immagine distorta, come si può immagine la causa è la natura continua della distorsione prodotta dalle lenti, mostrata in figura 3.2. Si cerca di sopperire a questa mancanza di corrispondenza perfetta stimando mediante una tecnica matematica chiamata interpolazione bilineare il valore approssimato dell'intensità del pixel in questione.

Tale tecnica è assai importante nel processo di rettificazione, non solo per il motivo di cui si è detto in precedenza, ma anche perché è alla base di uno degli algoritmi di rettificazione "semplificata", di cui si parlerà nei capitoli 3 e 4 di questa tesi.

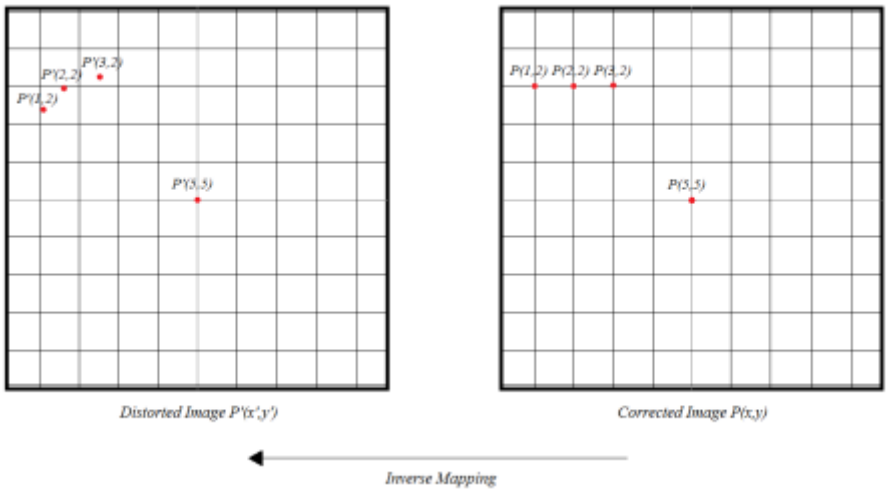


Figura 3.2 Problemi nel recupero pixel dell'immagine distorta: non c'è un pixel a ogni angolo della griglia che descrive l'immagine distorta, impossibile dunque pensare ad una corrispondenza realizzata con le sole tabelle del *displacement.* Il valore dell'intensità del pixel dell'immagine distorta, corrispondente a quello della immagine corretta, sarà stimato mediante interpolazione bilineare.

L'interpolazione bilineare, assunto che un pixel dell'immagine distorta difficilmente starà su uno degli angoli della griglia, userà l'intensità dei quattro pixel vicini, pesati opportunamente, al fine di determinarne l'intensità.

$$
\begin{aligned}
I^{rect}_{x_u,y_u} = {} & I^{raw}_{x_i,y_i}(1-x^f)(1-y^f) \\
& + I^{raw}_{x_{i+1},y_i}\, x^f(1-y^f) \\
& + I^{raw}_{x_i,y_{i+1}}\,(1-x^f)y^f \\
& + I^{raw}_{x_{i+1},y_{i+1}}\, x^f\, y^f
\end{aligned}
$$

Figura 4.1 Formula dell'interpolazione bilineare

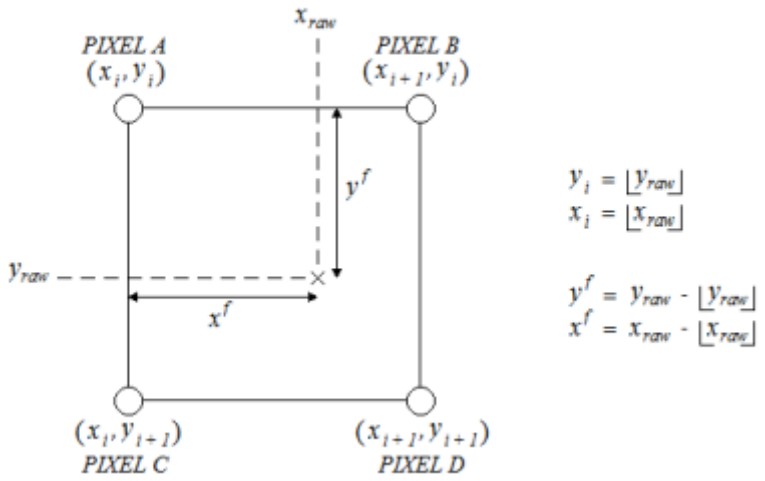


Figura 4.2 Xraw e Yraw sono le coordinate del punto della immagine distorta (ottenuto mediante tabelle del *displacements*). (Xi,Yi) rappresentano il pixel più vicino, ottenuto dalle coordinate (Xraw,Yraw) di cui si prende la parte intera. Infine (Xu,Yu) sono le coordinate del punto dell'immagine corretta di cui vogliamo trovare la corrispondente intensità nella distorta

3.3 Rettificazione "semplificata": implementazioni possibili

Nei capitoli precedenti si è evidenziato che la chiave per risolvere il problema della distorsione nelle immagini è la tabella del *displacements*; purtroppo però è proprio tale tabella a creare i maggiori problemi per quanto riguarda l'implementazione su dispositivi *embedded* come ad esempio una FPGA, poiché fa uso di:

- *Floating-point* per identificare l'entità dello scostamento tra i pixel dell'immagine corretta e quelli dell'immagine distorta;
- Grande uso della memoria dato che devono essere memorizzate ben 4 tabelle (una per l'immagine *left* per le x, un'altra per le y e altrettante per l'immagine *right*). Tali tabelle dipendono poi anche dalle dimensioni delle immagine quindi ad esempio per il *dataset* KITTI (risoluzione 1242x375) la cosa comporterebbe non pochi problemi.

3.4 Prima semplificazione: utilizzo dei *fixed-point*

Tipi di dato *floating-point* non sono utilizzabili in dispositivi come le FPGA al contrario

dei *fixed-point* visto che questi si basano sugli interi: i bit che rappresentano l'intero che caratterizza quest'ultimo tipo di dato saranno utilizzati in un certo numero per la parte decimale del valore della tabella del *displacement* e i rimanenti per la sua parte intera.
Scelti gli n-bit per la parte decimale si passa ai *fixed-point* moltiplicando il numero float per 2 elevato agli n-bit utilizzati, che equivale a fare uno *shift* di n-bit a sinistra e infine si deve effettuare poi un cast ad intero. L'inverso si ottiene facendo uno *shift* sempre di n-bit verso destra.
Utilizzare i *fixed-point* implica chiaramente una perdita di informazione, che sarà più o meno elevata in base alla disponibilità di bit per la parte decimale: infatti assegnare n bit per la parte decimale significa che ogni numero decimale sarà diviso in n parti (ad esempio se si hanno 3 bit per la parte decimale tra 0 e 1 ci saranno 8 intervalli e la precisione massima è da individuare nella dimensione di questi intervalli).
La perdita di informazione avverrà cioè quando si vuole passare da una codifica *float* a *fixed* e si vuole poi effettuare il processo inverso, cioè da *fixed* a *float*, che darà come risultato un *float* che però avrà la parte decimale non più uguale a quella che aveva in precedenza, ma uguale al valore di uno degli intervalli di divisione dei decimali di cui si diceva sopra.

3.5 Riduzione delle matrici di *displacements*: metodo *sampling*

I dispositivi come una FPGA impongono limiti in termini di occupazione e disponibilità di memoria. Pensare di salvare sul dispositivo tutte e quattro le matrici in maniera completa è difficilmente possibile: basti pensare che per una immagine con risoluzione 640x480 (cioè molto meno delle immagini Kitti utilizzate in questa tesi), se si sceglie di usare 16 bit per rappresentare i dati in *fixed-point* si dovrebbero utilizzare all'incirca 3 MB valore certamente non compatibile con la memoria interna (BRAM) della maggior parte delle FPGA attuali.
La soluzione a questo problema è rappresentata dal metodo *sampling*, sul quale tra l'altro si basa uno degli algoritmi utilizzati in questa tesi per la rettificazione delle immagini.

Le matrici vengono campionate ogni *sampling factor*: ad esempio, scegliendone uno con valore 8 per una immagine 640x480, ci sarà un campionamento ogni 80 colonne per la *width* e ogni 60 righe per la *height.*

Ridurre di un certo fattore ha un ottimo impatto sulla memoria ma una conseguente perdita di precisione: per ricostruire la tabella si utilizza ancora una volta la tecnica dell'interpolazione bilineare che introduce un errore quando dal punto dell'immagine corretta, considerando lo scostamento memorizzato nelle matrici, si va a posizionare *Xraw* e *Yraw* (cioè il corrispondente sulla distorta). Campionare implica inoltre anche un'operazione in più da fare per "ricreare" la matrice, ma nel mondo delle FPGA è più facile pensare a un calcolo in più piuttosto che ad un utilizzo massiccio della memoria.

L'utilizzo di una matrice ridotta chiaramente comporta un’operazione in più anche durante l'*inverse mapping*: infatti ora si dovrà non più prendere tutti i pixel dell'immagine corretta (cioè tutti gli angoli della griglia in cui tale immagine è suddivisa, come mostrato in Figura 3.2) e guardare il relativo scostamento, ma per ogni pixel si applicherà la divisione per il *samplingFactor* e si prenderà la parte intera del risultato: quella sarà la coordinata da usare per reperire lo scostamento delle matrici.

Un possibile problema del metodo *sampling* è quello che può essere definito "effetto di bordo": si pensi ad un'immagine 640x480 dove l'ultimo valore campionato con un *samplingFactor* ad esempio 8, fa si che per i pixel successivi a lui (cioè quelli che vanno da 632 a 639) non ci siano tutti e 4 i pixel per cui effettuare una interpolazione bilineare sufficientemente corretta: per questi si è soliti introdurre pixel neri, col risultato finale di avere una immagine rettificata caratterizzata da bordi neri sul lato destro e sul bordo inferiore, riducendo quindi l'area utile dell'immagine.

Vi sono soluzioni a questo problema:

- ricomporre completamente l'immagine al fine di avere area utile completa in maniera semplice e veloce. Per i bordi neri si utilizzerà infatti il valore dei pixel estremi dell'immagine, aggiungendoli alla matrice ridotta per consentire una interpolazione corretta: questa soluzione è solo apparentemente risolutiva, visto che introduce un errore tanto alto in quelle zone che le rende quasi inutilizzabili per altri scopi.

- idealmente si vorrebbe avere a disposizione il pixel immediatamente successivo all'ultimo della immagine in esame; per "ricostruirlo" è necessario considerare che vi sono relazioni matematiche ben definite tra i campioni delle matrici e ognuno di questi non è indipendente dagli altri, si deve quindi poter fare una stima in maniera intelligente per avere un pixel "in più" corretto. Studiando le matrici di *displacements* ci si è resi conto che si ha un aumento dei campioni in maniera progressiva procedendo verso il centro immagine da sinistra a destra e dunque ad esempio per stimare l'elemento in più di una riga possiamo prendere l'ultimo campione e sommarci la differenza tra questo e quello a lui precedente. Tale soluzione produce ad esempio un miglioramento in termini di errore medio dell'immagine ed è quindi da considerarsi la soluzione più corretta.

Alla luce di queste considerazioni si è compreso che si hanno due parametri da variare e testare al fine di capire quale *setting* consente un buon compromesso tra uso della memoria e qualità dell"immagine rettificata: il *sampling factor* e il numero di bit per la parte decimale del numero *float*, rappresentante lo scostamento, da approssimare col tipo di dato *fixed-point*.
In realtà emerge che tra i due il fattore predominante è il *sampling factor*: se si tiene "fermo" questo e si lascia variare il numero di bit per la parte decimale si nota che non si hanno grandi miglioramenti oltre un certo valore.

Capitolo 4. Implementazione del processo di rettificazione

Come si è anticipato nella sezione 3.2 in cui si era introdotto il concetto di *inverse mapping* l'idea della rettificazione passa attraverso il concetto di *remapping*, ovvero la definizione di una relazione tra le coordinate dei pixel di input e di quelli di output, in modo da risalire, utilizzando gli scostamenti, da pixel dell'immagine corretta a pixel dell'immagine distorta.
Il cuore della rettificazione è quindi condensato in due funzioni che OpenCV fornisce con la seguente firma (mostrata qui in C++):

- `void remap (InputArray src, OutputArray dst, InputArray map1, InputArray map2, int interpolation, int borderMode=BORDER CONSTANT, const Scalar& BorderValue=Scalar ( ))`
  dove i parametri rappresentano rispettivamente l'immagine sorgente (quella corretta), quella destinazione, gli scostamenti per le x, per le y, il tipo di interpolazione (la bilineare è di *default*).

Come si può notare la funzione `remap` utilizza le mappe di *displacements* e quindi presuppone che siano già ottenute dalla calibrazione. Per ottenerle OpenCV fornisce la funzione C++ `initUndistorRectifyMap` che si occupa, a partire dalle coordinate dei pixel dell'immagine corretta, di ottenere le coordinate di quelli della distorta. E' questa la funzione che si occupa di memorizzare in memoria le mappe di *displacements*.
In base alla disponibilità di risorse si deciderà che algoritmo utilizzare tra quelli che saranno mostrati in seguito: utilizzare un algoritmo piuttosto che un altro significa, in parte, decidere in che momento queste funzioni interverranno sulle immagini.

4.1 Distorsione delle immagini per testare i vari algoritmi

Come è stato anticipato, la valutazione è stata svolta sul dataset KITTI, che fornisce 200 immagini *left* e *right* perfettamente rettificate: per poter testare l'efficacia degli algoritmi di correzione delle distorsioni in tutte le varie forme, è necessario introdurre una distorsione artificiale.

E' in questo contesto che si inserisce il progetto realizzato da Paolo Di Febbo; tramite dei file di configurazione opportuni si simula il comportamento di cinque tipi di lenti diversi, da *lens0* fino a *lens4*, cioè da minima distorsione a massima distorsione.

Il processo di distorsione causerà la curvatura di linee che nella realtà dovrebbero essere curve e soprattutto porterà alcune parti dell'immagine al di fuori dei limiti dell'immagine stessa.

Figura 4.1 Immagine *left* originale

Figura 4.2 Utilizzo di *lens0:* distorsione praticamente non rilevabile ad occhio. Le linee stradali non sono curvate e i pali sono rimasti dei segmenti. Unico dettaglio che conferma una minima distorsione: il rosso del semaforo è meno visibile e i lampioni dietro l'auto bianca sono stati leggermente portati fuori dalle dimensioni dell'immagine

Figura 4.3 Utilizzo di *lens4:* distorsione elevata, i pali sono decisamente curvati, specialmente nel bordo immagine. Il rosso del semaforo e i lampioni dietro l'auto non sono nemmeno rilevabili

4.2 Rettificazione: algoritmo di *map sampling*

Il primo algoritmo di rettificazione "semplificata" è quello di cui si è parlato diffusamente in precedenza: quello con campionamento del valore delle matrici del *displacements* al fine di risparmiare l'uso della memoria.

Per via della considerazione fatta nel capitolo precedente in fase di test non è stato fatto variare il numero di bit da assegnare alla parte decimale per il passaggio *floating-point fixed-point* (fissato dunque ad 8) ma si farà variare solo il *sampling factor* tra 3, 5, 6 e 7 (intesi come esponenti per la base 2, quindi per esempio *sampling factor* 6 indica un campionamento ogni 64 valori della mappa di *displacements*).

Chiaramente, come già detto, la mappa va ricostruita il più precisamente possibile e l'idea più diffusa è quella di utilizzare l'interpolazione bilineare: ovviamente più il *sampling factor* aumenta più la ricostruzione della mappa diventa meno precisa e il processo di rettificazione ne risente. In realtà sarà dimostrato nel capitolo 6 "Risultati Sperimentali" che si può applicare una ingente riduzione della tabella senza allontanarsi troppo da quello che sarà definito come valore di riferimento.

Nonostante l'evidente risparmio di risorse in termini di memoria, che si paga con un sostenibile incremento dell'hardware per garantire il calcolo dell'interpolazione per le mappe, tale approccio non è utilizzato su larga scala forse per l'incertezza sull'effettiva precisione.

Per come è stato introdotto, per funzionare questo algoritmo ha bisogno di essere in possesso delle mappe di distorsione: solitamente vengono ottenute mediante l'utilizzo di determinate funzioni di OpenCV che forniscono un supporto alla calibrazione. Di consueto, tale operazione si svolge mediante l'utilizzo di una *chessboard*: la telecamera

deve individuarne i punti particolari (come gli angoli) e tramite una elaborazione mediante funzioni OpenCV in cui si verifica la corrispondenza fra punti dell'immagine e punti della scena reale si ottengono i parametri della telecamera, come la sua orientazione rispetto alla scacchiera, e i parametri intrinseci, come la distorsione introdotta dalle lenti. Tipicamente dunque queste mappe sarebbero ottenute prima di iniziare con le funzioni di rettificazione, si parlerebbe quindi di una fase di calibrazione *offline*.

All'interno di questo progetto non si segue questo flusso di esecuzione, poiché essendo la distorsione "artificiale", tali mappe sono generate a *run-time* a partire dalle dimensioni dell'immagine e da file di configurazione costruiti *ad hoc* per simulare le distorsioni.

In seguito sempre a *run-time* le mappe son passate alla funzione `remap` per ottenere le immagini rettificate.

Figura 4.3 Risultato della rettificazione con algoritmo *map sampling*, con fattore di riduzione delle matrici 6, su massima distorsione (utilizzo di *lens4*). Si nota come le linee della strada siano tornate regolari, come i pali non siano più curvati. Per quanto riguarda le regioni in nero e il contenuto informativo perso (come il rosso del semaforo) sarà fornita una spiegazione più avanti.

4.3 Rettificazione: algoritmo *on-the-fly computation*

Spesso si preferisce questa soluzione al *map sampling*, perché garantisce precisione maggiore nel determinare gli scostamenti.

L'idea di questo algoritmo, detto anche *on-the-fly map computation*, è quello di non utilizzare memoria (fatta eccezione per i parametri di configurazione, praticamente trascurabili in termini di spazio richiesto per la loro memorizzazione) e di effettuare tutti i calcoli, che normalmente vengono fatti *offline* per ottenere la tabella , in hardware.

Come calcoli si parla di un set di espressioni abbastanza complesso, che costituiscono una trasformazione, che in termini molto semplici visto che tale teoria non è il centro della tesi, consente di lavorare sulla immagine in input e di applicarci sopra una trasformazione che simuli la rotazione delle camere e dunque il loro allineamento, al fine di ottenere come già detto i pixel corrispondenti tra immagini *left-right* sulla stessa linea. Tale trasformazione è assai costosa, fa infatti ampio uso di moduli quali sommatori, moltiplicatori e soprattutto divisori, che sono quelli che veramente alzano il costo in termini di *hardware* (si intendono formule dunque che non utilizzano divisioni per potenze di due, realizzabili con un semplice *shift* a destra).
Dunque utilizzando l'algoritmo *computation* si risparmia sì memoria, ma si paga caro il fatto di realizzare in hardware operazioni complesse. Inoltre presenta altri due problemi:

- in hardware si è già precedentemente discusso sulla difficoltà di impiego di dati *floating-point*, preferendogli i *fixed-point*. La qualità della rettificazione sarà valutata in questa tesi dunque variano i bit da dedicare alla parte decimale (chiamati in questo contesto *fractional bits*), provando 12, 16, 20 bit.
- per ogni immagine acquisita dallo stesso sistema, nelle stesse condizioni, dunque con identici parametri di configurazione si ricalcola il tutto: grande ridondanza e grande costo in termini di risorse.

Nonostante i lati negativi tale sistema è ampiamente utilizzato, probabilmente per la precisione con cui si ottengono le mappe, nonostante l'introduzione dei *fixed-point.*

Figura 4.4 Risultato di rettificazione usando l'algoritmo *computing* con 20 *fractional bits* , su massima distorsione prodotta da *lens4*. Anche qui si notano le linee dritte della strada, ugualmente per i pali. Per il nero e i punti dell'immagine non presenti rispetto all'originale si faccia riferimento alle parti seguenti.

4.4 Valutazioni degli algoritmi di rettificazione

Per scegliere se un algoritmo è idoneo all'essere implementato su FPGA, sia dal punto di vista del risultato sia dal punto di vista delle risorse utilizzate, si hanno diversi parametri di valutazione.

Il lavoro di Paolo Di Febbo ha già analizzato questi algoritmi su un'altra tipologia di immagine, tenendo presenti tre metriche di valutazione:

- il costo delle risorse necessarie, in termini di hardware (sommatori, moltiplicatori, divisori, che variano in numero in base all'algoritmo scelto)
- errore geometrico, utilizzando *Root Mean Square Errore (RMSE),* che calcola l'errore quadratico medio tra i dati "corretti", cioè le mappe generate da OpenCV ovvero rettificazione "perfetta", e i dati "stimati", cioè le mappe ottenute usando gli algoritmi di rettificazione "semplificata".
- la disparità, ovvero vengono valutati come varia il calcolo della mappa di disparità utilizzando prima le immagini "corrette" e poi quelle su cui è stato svolto un lavoro di rettificazione. Questa tesi si pone l'obiettivo di ottenere un insieme di valori molto ampio che diano la possibilità di capire in modo robusto come un algoritmo di rettificazione piuttosto che un altro influisce nell'ottenere una disparità il più vicina possibile a quella di riferimento.

  Gli algoritmi di *stereo matching* utilizzati saranno discussi nel capitolo 5 e nel capitolo dei risultati sperimentali.

4.5 Utilizzo di maschere binarie

Sebbene l'utilizzo di queste maschere sia stato adottato in seguito ai primi risultati ottenuti con gli algoritmi di *stereo matching* saranno introdotte in questo capitolo visto che riguardano maggiormente la parte di distorsione delle immagini e solo in un secondo momento il calcolo della disparità.

Come si diceva in precedenza sono stati utilizzati file di configurazione che simulassero il comportamento di una lente: di conseguenza le immagini che venivano distorte avevano le parti di immagine vicine ai bordi che venivano "portate fuori" dai limiti. Come si può vedere dunque la distorsione provoca l'inserimento nell'immagine del nero che si vede maggiormente in figura 4.3 e che è amplificato nel momento della

rettificazione (figura 4.4). Quelle parti di immagine che dunque con l'utilizzo delle lenti vengono portate fuori al momento della correzione non si riesce ovviamente a riportarle dentro e quindi quella parte di immagine "non è più valida" e gli algoritmi di rettificazione la "marchiano" con pixel neri.
I primi test fatti con algoritmi di *stereo matching* effettuati sulle prime immagini rettificate davano risultati poco confortanti come errore medio totale per tutte e 200 le immagini: si alzava anche di 10 punti percentuali rispetto al riferimento. La cosa accadeva perché gli algoritmi *stereo* basano il calcolo di errore medio sul conteggio di quelli che nel prossimo capitolo vengono nominati come *pixel reali* e *pixel corretti*: il problema del nero è che questi *pixel* neri venivano considerati come buoni mentre non lo erano. Per evitare questo inconveniente sono state generate delle maschere binarie (figura 4.5); tali maschere hanno la stessa dimensione delle immagini KITTI e sono generate per ogni lente e sostanzialmente servono a dire, per ogni diversa distorsione, quali pixel considerare nel calcolo di quelli corretti e reali e quali no.
Si chiamano maschere binarie proprio per questo, hanno cioè due valori, codificati in base all'intensità del pixel:

- 255 (pixel bianco) dice che il pixel alla stessa posizione dell'immagine su cui si sta facendo *stereo matching* è da considerare come valido;
- 0 (pixel nero) significa l'esatto opposto;

In realtà poi nel calcolo dei pixel validi si cerca di essere conservativi e di considerare un possibile *aliasing* nei passaggi tra pixel bianchi e pixel neri e dunque si può impostare una soglia un po' più bassa di 255 (in questa tesi è stato scelto 250).
Come si vedrà nel prossimo capitolo l'utilizzo di queste maschere comporta una piccola modifica nel progetto Stereo Software.

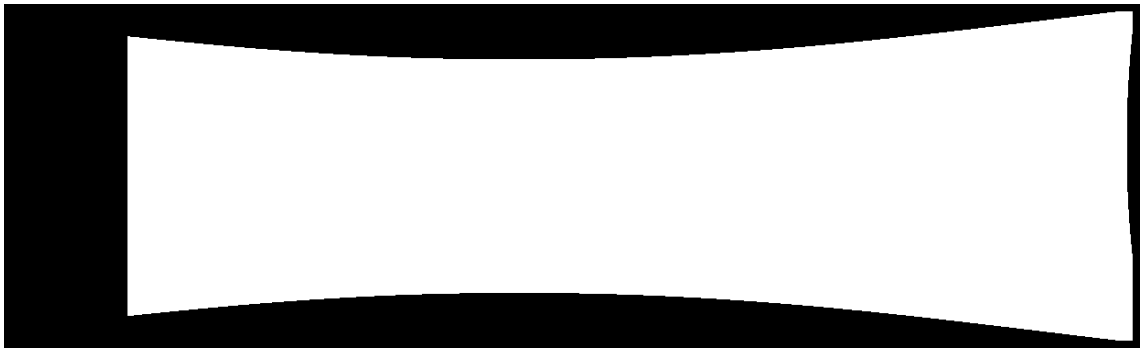


Figura 4.5 Esempio di maschera binaria per lens4: per distorsioni minori il bianco è molto più esteso e l'area utile è maggiore.

Figura 4.6 Mappa di disparità ricavata dalle immagini *left-right* in figura 4.4. Il calcolo sulla qualità della disparità sarà valutato sull'area che viene fuori sovrapponendo, idealmente, la maschera di figura 4.5

Capitolo 5. Problema della corrispondenza stereo

Si specifica che ci sono vari algoritmi per questo ambito, nelle più svariate forme: tuttavia la loro teoria non è oggetto di questa tesi e meriterebbe una trattazione a parte e per questo motivo saranno qui illustrati solo quelli che sono stati utilizzati in questa tesi.
La scelta degli algoritmi *stereo* non è stata casuale, sono stati infatti scelti due algoritmi:

- quello che fornisce l'errore medio minore tra quelli offerti dal progetto Stereo Software, dove per errore medio totale si intende la media ponderata tra l'errore percentuale per ogni immagine e il numero di pixel *reali*
- quello denominato *block matching*, che è stato scelto per dare continuità col lavoro svolto da Paolo Di Febbo, al fine di estendere un insieme di risultati già esistente

5.1 Come ottenere il calcolo della disparità

Come si è già detto la funzione della rettificazione è quella di correggere le immagini in modo che la ricerca dei punti corrispondenti tra *left e right* avvenga su una sola linea, cioè in 1D.
Gli algoritmi *stereo* sostanzialmente calcolano per ogni *pixel* nell'immagine un valore che è inversamente proporzionale alla distanza tra il punto nella scena reale e il sistema *stereo* che lo rileva. Come si può notare dalla figura 1.3 vi è una importante relazione fra le telecamere e il punto della scena: per ogni punto dell'immagine *left*, il corrispondente sulla *right* ha un valore per la coordinata x che minore o al più uguale al valore della x della *left.* Questa considerazione è importante perché vincola la ricerca non su tutta la riga ma su una finestra più ristretta; se si vuole vincolare ancora di più, come avviene nella realtà, impostando un valore massimo di disparità allora la ricerca avviene all'interno di una finestra ancor più ridotta: viene definito un *range* [DISPARITY_MIN, DISPARITY_MAX] e la ricerca sarà effettuata quindi dalle stesse coordinate del punto dell'immagine di riferimento, spostandosi a sinistra di un valore al massimo DISPARITY MAX. Naturalmente riducendo DISPARITY MIN la disparità per punti lontani aumenta in precisione, aumentando DISPARITY MAX avviene lo

stesso ma per i punti vicini. Definito lo spazio in cui eseguire la ricerca si deve definire il metodo di confronto, che per forza di sarà l'intensità del pixel, unica informazione disponibile.

Il modo più semplice per identificare il pixel corrispondente è quello di trovare nel *range* scelto quello che minimizza la differenza delle intensità (si definisce così il *matching cost*): si parla in questo caso di strategia *WINNER TAKES ALL* (figura 5.1).

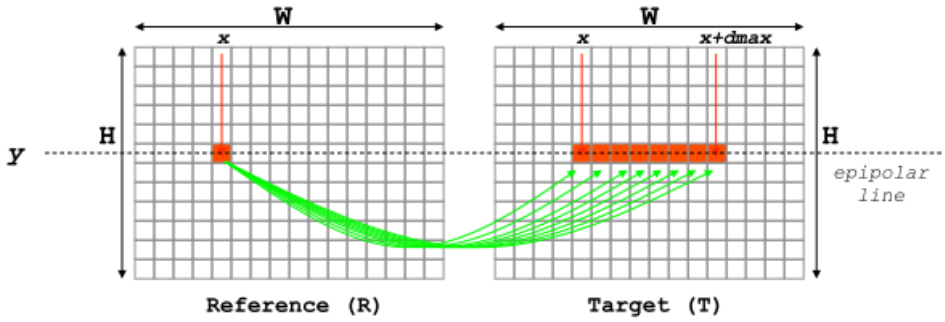


Figura 5.1 Metodo più semplice per trovare il pixel corrispondente tra le immagini

Questo tipo di algoritmi sono detti *locali* perché la ricerca è estesa alle zone vicine al punto in esame: sono tecniche buone dal punto di vista delle risorse e delle *performance*, meno per quanto riguarda la qualità della mappa di disparità che danno in *output*, soprattutto in presenza di regioni uniformi o occlusioni. Alcuni algoritmi locali cercano di aggregare i costi in una finestra di dimensione prestabilita (quindi si fa la somma di tutti i valori di intensità dei pixel appartenenti alla finestra il cui centro è il pixel in esame e si confrontano le finestre con l'immagine *right*), cosa che però comporta una perdita in precisione visto che i bordi e quindi parti differenti vengono calcolate tutte assieme.

Vi sono anche altri algoritmi, detti *globali*, che cercano il corrispondente del pixel in esame non solo nelle vicinanze ma per tutta l'immagine: si va alla ricerca di punti che minimizzino una determinata funzione di costo. Una loro variante sono i *semi-globali*, che ricercano non su tutta l'immagine ma solo in determinate parti di questa.

In questa tesi saranno utilizzati:

- per gli algoritmi locali la trasformata *census* in due delle sue forme che saranno spiegate poco più avanti (binaria a griglia pari e binaria a griglia completa, detta

anche *Block Matching)*; poi sarà applicato il *box filtering*, una tecnica di ottimizzazione per l'aggregazione dei costi

- per gli algoritmi semi-globali sarà utilizzato l'algoritmo SGM classico (che prende in input ciò che viene elaborato dalla census per ottenere il costo puntuale e su cui poi viene fatta l'aggregazione dei costi, utilizzando anche qui un meccanismo in stile *pipeline*)

La trasformata census sfrutta una finestra di pixel il cui centro è il pixel in esame e produce una stringa di bit (solitamente la finestra si sceglie di dimensione N quadrata, quindi la stringa sarà di N bit e nel caso della tesi di dimensione 5) che avrà tanti uni quanti sono i pixel della finestra maggiori per intensità al pixel centrale, altrimenti zeri.
Nonostante la census abbia un costo moderato visto che i confronti possono essere fatti in parallelo ci sono versioni alternative a quella standard in cui i confronti vengono fatti a scacchiera, perdendo sì informazione, ma diminuendo la lunghezza della stringa di bit.
Per il calcolo dei costi locali bisogna fare un confronto tra le stringhe di bit *left e right:* si usa a tal fine il metodo Hamming, che fa un confronto bit a bit in base alla posizione che questo occupa all'interno della stringa e restituisce il numero delle posizioni in cui si hanno bit diversi.
Il passo successivo è quello di aggregare i costi usando una finestra come si diceva in precedenza, che però presenta i problemi di cui si è detto sopra; nonostante ciò, visto il poco costo di questa tecnica, se ne fa un largo utilizzo “nel mondo reale”. Si cerca di migliorare il risultato utilizzando la tecnica del *box filtering.*
Per il calcolo della disparità col metodo *Block Matching* ci si ferma a questo stadio, mentre in questa tesi è stato eseguito anche il passo successivo, ovvero l'utilizzo di SGM, il quale al posto che lavorare direttamente sui pixel lavora sul risultato prodotto dagli stadi fino al *box filtering*. SGM lavora su otto *scanline*, ovvero otto percorsi spaziati di 45° tra loro e su questi percorsi si va alla ricerca del pixel che minimizza una determinata funzione di costo, che cerca di essere più accurata del semplice confronto pixel-pixel e soprattutto cerca di risolvere il problema per le regione uniformi e per quelle di discontinuità come i bordi, le superfici inclinate e quelle curve, con l'obiettivo di rendere graduale la disparità. Le funzioni di costo saranno poi aggregato lungo i percorsi e ne sarà calcolato il minimo, che sarà il candidato per il calcolo della disparità.

Questo è un algoritmo molto portabile su FPGA visto che lavora con soli interi e riesce comunque ad ottenere mappe di disparità precise, spendendo però in termini di risorse di calcolo.

5.2 Modifiche allo Stereo Software

Per calcolare la disparità si fa uso di una struttura dati 3D (una matrice 3D) denominata Disparity Space Image (DSI) dove ogni elemento della matrice rappresenta il costo della corrispondenza tra l'intensità del pixel dell'immagine *left* e quella dell'immagine *right.* Vengono cioè salvati tutti i costi (le disparità) per ogni coordinata e si genera una mappa a partire da questa struttura.

Ora è il momento di valutare la correttezza della mappa generata: bisogna calcolarne l'errore medio rapportando i pixel "corretti" e quelli totali per ogni immagine.

Per farlo si usa un'immagine di riferimento, come spiegato nella tesi di Andrea Bombino, detta *ground-truth*, che ottiene i valori corretti per i pixel dell'immagine e per farlo utilizza un laser e un sensore *Lidar*: per ogni immagine però non si riesce ad ottenere veramente un valore per tutti i punti e quindi questi pixel non identificati saranno messi a zero, mentre quelli rilevabili vengono definiti in questa tesi come *pixel reali*, nome che intuitivamente rende l'idea. Inoltre tra i pixel reali solo alcuni sono privi di errore e questi, sempre per averne una idea intuitiva, vengono definiti *pixel corretti*.

Per definire se un pixel è "corretto" dovrà necessariamente essere "reale" e inoltre dovrà soddisfare una condizione che relazione sia la *ground-truth* sia la *disparity-map;* per essere "reale" invece per gli algoritmi *stereo* su KITTI standard deve avere valore diverso da zero nella immagine di *ground-truth.* Con l'utilizzo delle immagini rettificate invece si introduce una modifica da apportare al codice che si basa sulla teoria di cui si è parlato al punto 4.5: per essere considerato reale il pixel dovrà essere sottoposto al controllo della maschera binaria. Per implementare ciò sarà data in input agli algoritmi *stereo* oltre le immagini KITTI, la mappa e la *ground-truth* anche tale maschera.

```
long PIXEL_TOTAL = 0, REAL_PIXEL = 0, PIXEL_CORRECT = 0;
for (int k = 0; k < hL; k++) {
    for (int j = 0; j < wL; j++) {
        if (CV_IMAGE_ELEM(G, ushort, k, j) != 0 && CV_IMAGE_ELEM(MASK_LENS, ushort, k, j) >= 250) {

            if ((abs((CV_IMAGE_ELEM(G, ushort, k, j) / 256) - CV_IMAGE_ELEM(DisparityL, uchar, k, j))) <= 3) {
                PIXEL_CORRECT++;
            }
            REAL_PIXEL++;
        }
        PIXEL_TOTAL++;
    }
}
```

Figura 5.2 Codice in cui si fa il conteggio dei pixel reali e corretti per il calcolo dell'errore medio. L'utilizzo delle maschere si nota nel primo IF e comporta l'inserimento della seconda condizione

Capitolo 6. Risultati sperimentali

Per calcolare l'errore medio totale ottenuto con gli algoritmi *stereo*, in particolare per verificare quanto un determinato algoritmo di rettificazione impatta sulla qualità della *stereo correspondence*, sarà calcolato per ogni immagine l'errore percentuale e il numero di pixel "reali", sarà poi fatto il rapporto tra la somma di tutti i pixel "reali" e la somma di tutti gli errori. Il tutto è stato realizzato mediante l'uso di file in formato ".csv".

Riepilogando quanto detto nei capitoli precedenti la sperimentazione è avvenuta su due tipi di algoritmi di rettificazione: *map sampling* e *on-the-fly map computation.*
Per quanto riguarda la verifica della disparità, riepilogando, sono invece stati utilizzati due algoritmi *stereo: block matching* (ovvero trasformata census a griglia completa con aggregazione dei costi fornita dal *Box Filtering*) e quella che può essere definita come una *pipeline* composta da più stadi (e che in seguito, per brevità, sarà così riferita), in cui viene prima calcolata la trasformata census a griglia binaria, si usa *box filtering* con dimensione della finestra pari a 5, poi interviene SGM classico (8 *scanline*, cioè tutte, settando il parametro MASK a 255). Le penalità P1 e P2, di cui non si è discusso ma che intervengono laddove siano usate funzioni di costo che vogliono evitare variazioni brusche di disparità, sono state impostate rispettivamente a 2 e a 30.
Prima di verificare l'impatto degli algoritmi di rettificazione bisogna definire quello che è il riferimento per il calcolo della disparità (cioè algoritmi *stereo* applicati a immagini KITTI standard): con l'algoritmo di *stereo matching pipeline* si arriva a un errore medio totale di 7.34 %. Per correttezza è opportuno però utilizzare le maschere anche quando si calcola il riferimento, in modo da evitare di calcolare la disparità per punti di cui poi non si dispone nelle immagini rettificate (cioè quelli che potevano essere stati portati fuori col processo di distorsione), al di là che questi punti possano alzare o abbassare l'errore medio totale. Per essere sicuri che con l'utilizzo delle maschere non si va a modificare il "significato" delle immagini si può confrontare il riferimento ottenuto con *lens0* e ciò che è rettificato partendo da una distorsione prodotta da *lens0:* dovrebbero assomigliarsi molto e la cosa è confermata come si vede sotto.

6.1 Algoritmo di rettificazione *map sampling*

All'interno delle tabelle sono mostrati gli errori calcolati sulle 200 immagini KITTI per ogni diversa configurazione, cioè per ogni variazione delle lenti di distorsione e per ogni variazione del *sampling factor*. Tali errori sono da considerarsi in percentuale e approssimati alla seconda cifra decimale.

Algoritmo *stereo: pipeline* (enunciata sopra)

LENS	sampling factor 3	sampling factor 5	sampling factor 6	sampling factor 7	riferimento
0	5,85	6	6,41	9,21	5,88
1	6,11	6,25	6,65	10,14	6,01
2	6,52	6,5	6,78	10,29	6,09
3	7,05	7,14	7,38	11,08	6,16
4	7,94	8,17	8,65	12,9	6,24

Si mostra ora un *plot* dei valori riportati in tabella in modo da poterne avere una idea di insieme e capire quale algoritmo consente di ottenere il miglior rapporto qualità-costo.

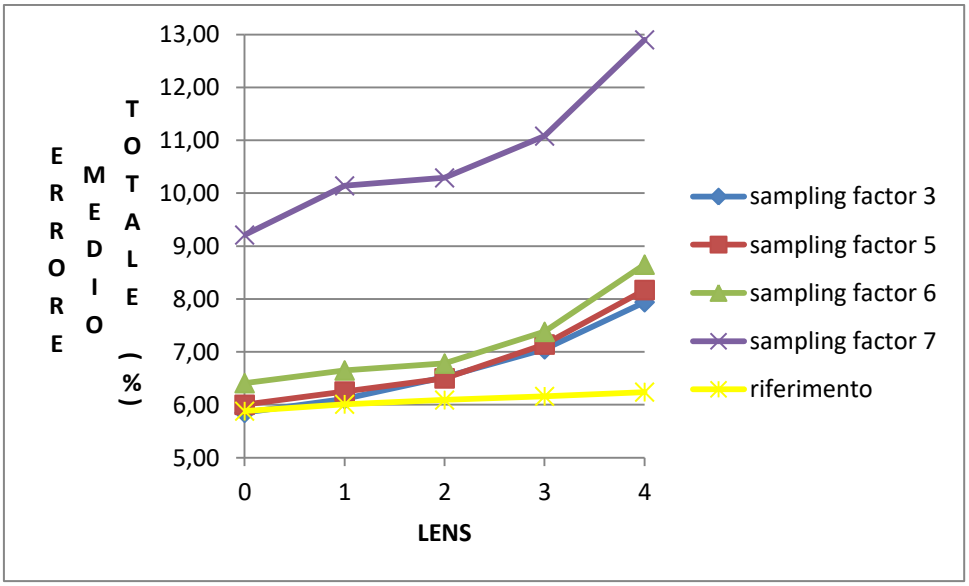


Figura 6.2 Come si nota dal plot *sampling factor* 3,5,6 danno ottimi risultati infatti non vanno mai oltre il 3% al riferimento e anzi fino a lens2 la differenza è difficilmente percettibile.

Come notiamo l'andamento è sempre monotono crescente e la cosa è in linea con l'idea che aumentando la distorsione aumenti anche l'errore medio degli algoritmi *stereo*. Per

quanto riguarda la valutazione della rettificazione si denota come con *sampling factor* 7 si sia ampliato notevolmente l'intervallo di campionamento. Tuttavia, anche in questo caso il risultato, sebbene affetto da un errore elevato, potrebbe essere accettabile in casi in cui si richiede una memorizzazione di una mappa molto ridotta.

Al contrario si può sfruttare un *sampling factor 6* avendo un andamento che non si distacca troppo da quello ottenuto con *sampling factor 3*, che è un po' troppo ottimista in termini di occupazione della memoria e che ovviamente garantisce ottimi risultati.

Sampling factor 5 fornisce un ottimo risultato ma il rapporto migliore in termini di qualità-costo lo raggiunge il valore 6, visto che non si allontana mai più di 3 punti percentuali dal riferimento e consente comunque una forte riduzione della matrice.

Algoritmo *stereo: block matching*

LENS	sampling factor 3	sampling factor 5	sampling factor 6	sampling factor 7	riferimento
0	26,62	27,01	27,4	29,14	28,82
1	27,16	27,45	28,38	34,16	27,73
2	27,41	27,6	28,25	33,53	28,68
3	27,93	28,12	28,55	33,38	28,67
4	28,67	28,92	29,47	34,12	28,74

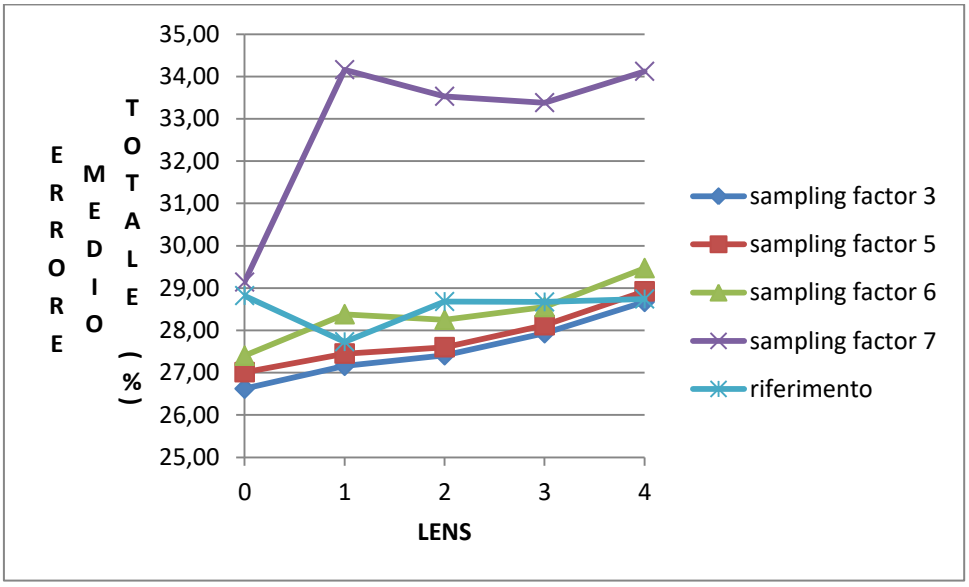


Figura 6.2 Errore medio nel calcolo della disparità variando i *sampling factor*. Abbastanza coerente l'aumento dell'errore medio aumentano il *sampling factor*. Corretto che l'errore medio si alzi aumentando la distorsione delle lenti.

Il plot mette in evidenza le caratteristiche dell'algoritmo *block matching* e degli algoritmi locali in genere, l'errore medio infatti esplode rispetto a prima. Le immagini KITTI sono particolarmente problematiche per questi algoritmi, i cui risultati mettono in luce tutti i problemi che si affrontano in casi di occlusione, dove cioè in presenza dei bordi degli oggetti abbiamo regioni che sono viste in una soltanto delle immagini *stereo,* oppure altri problemi sorgono in casi di regioni uniformi, come nell'esempio di figura 6.3.

Figura 6.3 Una immagine come questa con algoritmi locali potrebbe essere problematica nelle regioni a tessitura uniforme.

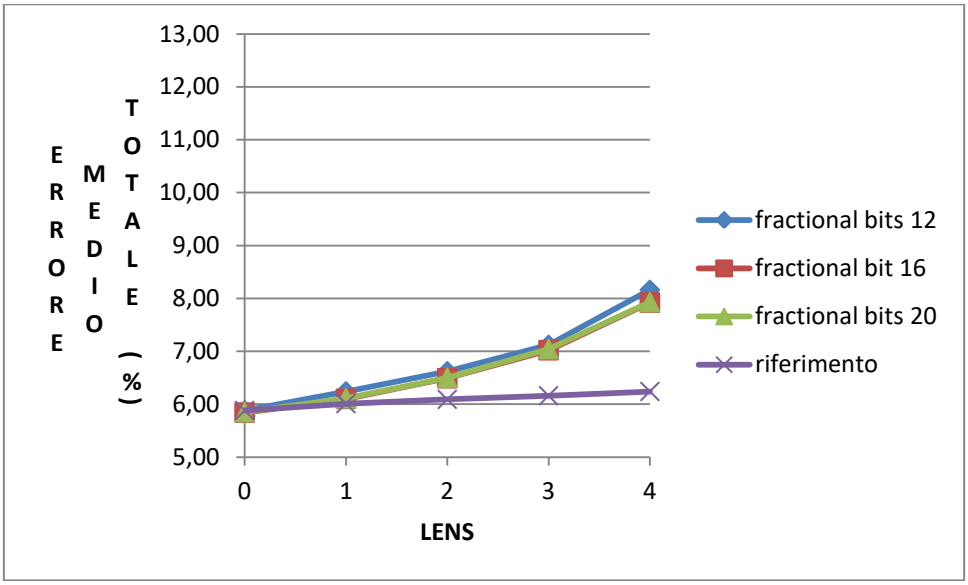
Come risultato positivo però vi è che gli algoritmi di rettificazione funzionano in maniera corretta, per tutti i parametri, 7 escluso, dove l'errore medio si alza eccessivamente. Infatti, abbiamo un andamento che non si discosta troppo dal riferimento, anzi in alcuni casi abbassa anche l'errore medio totale ma la cosa è da considerarsi casuale e forse causata dall'utilizzo delle maschere che vanno a modificare un po' il calcolo dell'area utile per l'ottenimento dell'errore medio totale. Il risultato quindi è positivo per quanto riguarda la valutazione dell'impatto degli algoritmi di rettificazione semplificata sullo *stereo matching*.

6.2 Algoritmo di rettificazione *on-the-fly map computation*

All'interno delle tabelle sono mostrati gli errori calcolati sulle 200 immagini KITTI per ogni diversa configurazione, cioè per ogni variazione delle lenti di distorsione e per ogni variazione dei *fractional bits.* Tali errori sono da considerarsi in percentuale e approssimati alla seconda cifra decimale.

Algoritmo *stereo: pipeline*

LENS	fractional bits 12	fractional bits 16	fractional bits 20	riferimento
0	5,88	5,84	5,85	5,88
1	6,24	6,1	6,11	6,01
2	6,62	6,49	6,5	6,09
3	7,12	7,02	7,04	6,16
4	8,16	7,92	7,93	6,24



I risultati con questa tecnica sono ottimi e fino a *lens2* non si notano grandi differenze tra il riferimento e il resto. In particolare si nota come variando i *fractional bits* non si ottengano grandi differenze in termini di errore medio e si può dunque tendere a risparmiare il più possibile.

Algoritmo *stereo: block matching*

LENS	fractional bits 12	fractional bits 16	fractional bits 20	riferimento
0	27,04	26,49	26,61	28,82
1	28,36	27,13	27,16	28,73
2	28,47	27,37	27,39	28,68
3	28,66	27,89	27,93	28,67
4	29,36	28,64	28,68	28,74

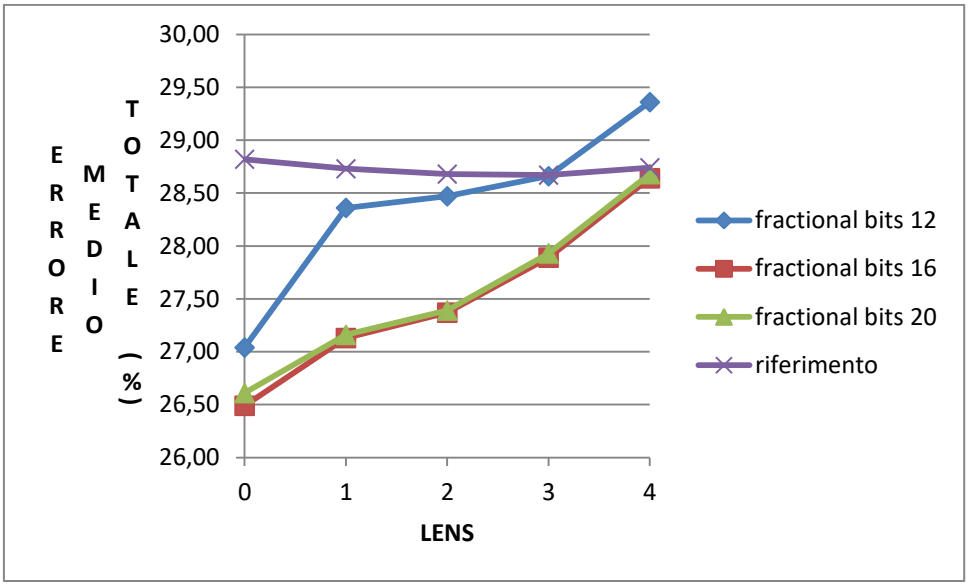


Figura 6.4 Approccio *computation* conferma ottimi risultati in caso di *fractional bits 16* e 20 ma anche in caso di 12 bit si è sempre negli intorni del riferimento

Il plot non porta grandi novità rispetto a quanto ottenuto con l'algoritmo di rettificazione *map sampling*. La rettificazione è leggermente migliore ma non di molto. Confermati, come prevedibile, i maggiori errori prodotti dall'algoritmo *block matching*.

Capitolo 7. Conclusioni

7.1 Considerazioni finali sui dati ottenuti

Analizzando più nel dettaglio i risultati mostrati nel capitolo 6 si deduce che, nel corso della sperimentazione effettuata, gli algoritmi di rettificazione non impattano in maniera eccessivamente negativa sul calcolo della corrispondenza tra i punti nelle due immagini. In particolare, se non si eccede con parametri di campionamento eccessivi (e.g., nel caso della sperimentazione eseguita, per esempio, con *sampling factor 7)*, si ottengono risultati più che soddisfacenti utilizzando *map sampling* con valore di campionamento pari a 6 e anche ottimi risultati con il metodo di rettificazione *on-the-fly computation* utilizzando 16 bit per la parte decimale.

Se si scelgono queste due configurazioni allora l'errore medio totale nel calcolo della *disparity* non si discosterà eccessivamente da quello di riferimento.

Volendo fare un confronto tra le due metriche di rettificazione si può guardare con fiducia al metodo *map sampling*, nonostante il suo scarso utilizzo in situazioni reali: il suo errore medio con la giusta configurazione non si discosta dal metodo *computation*, che certamente si conferma una garanzia per ottenere le migliori prestazioni anche se non sempre compatibile con esigenze di implementazione su sistemi embedded e in particolare su FPGA.

7.2 Considerazioni e conclusioni personali sul lavoro svolto

Dal punto di vista dei risultati è stato raggiunto l'obiettivo proposto, ovvero quello di realizzare un ampio set di risultati che mette a confronto le varie metodologie di rettificazione e il relativo impatto sugli algoritmi *stereo,* e quello di creare una grande raccolta di immagini rettificate coi diversi algoritmi, con tutte le possibili combinazioni di parametri.

Dal punto di vista personale sicuramente questa tesi ha contribuito ad ampliare le conoscenze sia in termini di linguaggi (Python non era mai stato studiato) che in termini di concetti teorici non affrontati nel ciclo delle lezioni (tutta la parte di visione stereo era nuova).

Importante e nuovo per me è stato anche il fatto di non costruire tutto il lavoro da zero ma di dover interagire con progetti già realizzati da altre persone, di doverci "mettere le mani" e muoverci i primi passi in autonomia.
Inoltre posso dirmi soddisfatto per quanto riguarda l'ambito scelto: dopo il tirocinio questo è il secondo passo nel grande mondo della computer vision (seppur ne abbia visto una parte più che relativa) e penso che potrà essere uno degli indirizzi dove investire i prossimi periodi di studio.

7.3 Sviluppi futuri

Il progetto così impostato consente anche ulteriori sviluppi futuri: nell'ambito di questa tesi è stato sviluppato l'impatto che avevano gli algoritmi di rettificazione di cui si è detto sopra su due algoritmi *stereo*, scegliendo quello che minimizzava l'errore medio e quello detto *block matching* per una questione di compatibilità con un set di risultati già presente che con il lavoro svolta poteva essere ampliato.

Dato che il progetto Stereo Software mette a disposizione numerose altre combinazioni di algoritmi per lo *stereo matching* vale dunque la pena verificare come impatta la rettificazione su questi: potrebbero trovarsi novità interessanti come ad esempio configurazioni particolari della trasformata census o varianti di SGM che per immagini rettificate in un determinato modo abbassano l’errore medio.

8. Bibliografia

[1] S. Mattoccia, “Stereo Vision: Algorithms and Application”, 2011, http://vision.disi.unibo.it/~smatt/

[2] L. Di Stefano, *http://didattica.arces.unibo.it/file.php/59/Elaborazione_dellImmagine_L-S/CVIP/2_ImageFormationAndAcquistion.pdf*

[3] *Rectification and Disparity*, *http://campar.in.tum.de/twiki/pub/Chair/TeachingWs09Cv2/3D_CV2_WS_2009_Rectification_Disparity.pdf*

[4] Tesi di Vincenzo Villani, “Algoritmi per la correzione di distorsioni in immagini”, Tesi di laurea, Università di Bologna AA 2015/2016

[5] Dataset KITTI 2015, *http://www.cvlibs.net/datasets/kitti/*

[6] OpenCV, *http://docs.opencv.org/2.4.11/*

[7] Andrea Bombino, “Valutazione sperimentale di algoritmi di stereo visione finalizzati ad una successiva implementazione su FPGA”, Tesi di laurea, Università di Bologna AA 2015/2016

[8] Paolo Di Febbo, “Real-time 3D sensing on FPGA”, Tesi di laurea, Università di Bologna AA 2015/2016

[9] Dataset Middlebury, *http://vision.middlebury.edu/stereo/data/*

Ringraziamenti

Ringrazio il Professor Stefano Mattoccia per la disponibilità e la fiducia riposta in me nell'assegnarmi questo compito.

Ringrazio i correlatori Matteo Poggi e Paolo Di Febbo: senza il loro supporto, la loro conoscenza e la loro grande disponibilità in ogni momento non sarebbe stato possibile giungere alla fine di questo percorso.

Ringrazio le mie colonne portanti: i miei genitori per i sacrifici fatti per permettermi di studiare nel migliore dei modi e per avermi fatto credere in me stesso, mia sorella per il grande aiuto, che mi ha permesso di dedicare questa estate praticamente solo allo studio liberandomi dal lavoro, la mia ragazza per avermi sopportato anche quando non era più umanamente possibile sopportare il mio stress e per essermi stata sempre vicino.

Ringrazio i miei nonni, che mi hanno sempre reso fiero di quello che facevo, spingendomi a dare il meglio.

Ringrazio i miei amici di Cesenatico per i momenti di svago.

Ringrazio tutti i miei amici dell'Università coi quali ho condiviso tutto in questi 3 anni: senza di loro sarebbe stato tutto più difficile e averli conosciuti ha dato grande valore a questi 3 anni (un grazie speciale ad Andrea per la disponibilità ad aiutarmi con la tesi).

Ringrazio tutti coloro che mi hanno aiutato in qualche modo e che hanno contribuito a questo risultato.