ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

FACOLTÀ DI INGEGNERIA

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

DIPARTIMENTO DI ELETTRONICA, INFORMATICA E SISTEMISTICA

TESI DI LAUREA

in

Elaborazione dell'immagine M

RICOSTRUZIONE 3D DA IMMAGINI RGB-D SU PIATTAFORMA MOBILE

CANDIDATO:
Nicholas Brunetto

RELATORE:
Chiar.mo Prof. Ing. **Luigi Di Stefano**

CORRELATORE:
Dott. Ing. **Nicola Fioraio**

Anno Accademico 2014/2015

Sessione I

Sommario

Introduzione

I progressi compiuti di recente in ambito tecnologico hanno portato ad una ampia diffusione dei dispositivi mobili, quali tablet e smartphone. Le capacità computazionali di tali dispositivi crescono con notevole rapidità, nel tentativo di eguagliare le performance tipiche dei computer desktop, senza rinunciare ad un elevato grado di mobilità. Tutto ciò ha portato allo sviluppo di nuove applicazioni in grado di sfruttare appieno le caratteristiche peculiari dei dispositivi mobili. Questo lavoro di tesi si focalizzerà su un particolare utilizzo di tali dispositivi, allo scopo di realizzare un'applicazione che consenta la ricostruzione 3D in tempo reale degli ambienti visitati.

L'applicazione realizzata propone una soluzione, su piattaforma mobile, al problema noto in letteratura come Simultaneous Localization And Mapping (SLAM), ovvero l'esplorazione di un ambiente sconosciuto da parte di un dispositivo, al fine di ricostruirne una mappa ed allo stesso tempo localizzare il dispositivo stesso. I diversi approcci proposti negli anni hanno visto l'utilizzo di sensori di vario genere, aventi lo scopo di raccogliere informazioni dall'ambiente circostante, efficacemente combinate al fine di risolvere il problema. Un esempio dei sensori più comunemente utilizzati è dato da: scanner laser, camere, GPS e IMU. Questo lavoro, in particolare, si colloca nel campo del Visual SLAM, sfruttando esclusivamente informazioni di tipo visivo.

Nel passato, i sistemi SLAM sono stati prevalentemente utilizzati su robot o veicoli autonomi e richiedevano risorse computazionali molto elevate ed un hardware avanzato e costoso. Lo sviluppo tecnologico da un lato, mettendo a disposizione hardware sempre più performante a basso costo, e la ricerca dall'altro, con nuovi approcci computazionalmente sempre più leggeri,

hanno infine reso possibile l'implementazione di un algoritmo SLAM su una piattaforma mobile. È questo il caso di SlamDunk, algoritmo SLAM in grado di lavorare con sensori RGB-D[1] di basso costo e facile utilizzo in diversi contesti applicativi, come ad esempio il Microsoft Kinect e l'Asus Xtion Pro Live. Tale algoritmo, attualmente implementato in ambiente desktop, è in grado di sostenere l'elevato frame rate dei sensori utilizzati, eseguendo in real-time a 30 frame al secondo, senza rinunciare alla qualità della ricostruzione generata. L'output dell'algoritmo è rappresentato come di una mappa 3D dell'ambiente esplorato, permettendo un successivo utilizzo della stessa in diversi campi applicativi, dalla navigazione all'analisi semantica. Lo scopo principale di questa tesi è il porting dell'algoritmo in ambiente mobile, permettendo così una ricostruzione 3D su hardware a basso costo e di facile reperibilità. Il porting da realizzare dovrà tenere in considerazione le limitazioni presenti in ambito mobile, come ad esempio un utilizzo ridotto della memoria rispetto ai computer desktop ed una potenza di calcolo lontana dall'essere paragonabile agli stessi. Si dovrà perciò tenere conto di tali problemi e cercare la soluzione migliore per limitarne l'influenza sull'applicazione finale. La piattaforma di riferimento per questo lavoro è un tablet con sistema operativo Android.

Le caratteristiche innovative di tale applicazione la collocano nell'emergente campo di ricerca sui dispositivi mobili, ponendola come reale alternativa ai classici approcci di risoluzione del problema SLAM in ambito desktop.

La tesi è stata organizzata in diversi capitoli. Il Capitolo 1 presenta in maniera approfondita il problema SLAM, fornendo una panoramica dei principali metodi di risoluzione, sia dal punto di vista algoritmico, sia dal punto di vista hardware. Il Capitolo 2 descrive SlamDunk, fornendo i dettagli implementativi dell'algoritmo e paragonandolo sinteticamente ad altre soluzioni presenti in letteratura. Il Capitolo 3 presenta alcune

[1] I sensori RGB-D rappresentano le informazioni da una parte attraverso un'immagine RGB, tipica delle classiche videocamere, e dall'altra attraverso l'indicazione della distanza a cui si trova ogni pixel dell'immagine. Essi funzionano adottando una videocamera ed un sensore ad infrarossi in grado di misurare la profondità.

importanti considerazioni relative al porting di SlamDunk su piattaforma mobile, sottolineando le problematiche e le relative soluzioni che è possibile adottare. Il Capitolo 4 si concentra sulla realizzazione del porting, sottolineandone le caratteristiche e le scelte progettuali intraprese. Il Capitolo 5, infine, presenta i risultati sperimentali in diversi contesti applicativi, assieme ad uno studio del comportamento dell'algoritmo al variare dei suoi parametri principali. Chiudono alcune considerazioni in termini di qualità e performance.

Capitolo 1
SLAM

Come punto iniziale di questo lavoro di tesi, risulta necessario un approfondimento del problema noto come Simultaneous Localization And Mapping, in modo tale da comprenderne a fondo l'utilità nonchè le possibili applicazioni.

Si partirà dalla definizione generale di SLAM, per poi passare ad una panoramica delle principali soluzioni a questo problema, suddivise in base alla tipologia. Sono perciò presenti soluzioni che si differenziano dal punto di vista dell'algoritmo utilizzato e soluzioni che si differenziano in base al tipo di sensore adoperato per la raccolta dei dati. Questa differenziazione è fondamentale in quanto la ricerca sul problema si è frammentata negli anni, percorrendo strade anche molto differenti fra di loro.

Di ogni specifica soluzione sono evidenziati pregi e difetti, in modo tale da individuare quelle che meglio si prestano ad una implementazione su piattaforma mobile.

1.1 Definizione

Per Simultaneous Localization And Mapping (SLAM) si intende un problema caratterizzato da due operazioni principali [KM07]:

- *Mapping*: Costruzione di una mappa dell'ambiente circostante. Solitamente la costruzione avviene in seguito alla raccolta di dati da sensori presenti all'interno del dispositivo, in grado di fornire informazioni utili sul luogo che si sta correntemente visitando;
- *Localization*: Calcolo della posa, ovvero posizione ed orientamento del dispositivo, una volta conosciuto l'ambiente nel quale lo stesso si muove. È solitamente necessaria la disponibilità di una mappa del luogo per poter comprendere la posizione del dispositivo all'interno della stessa.

Il problema SLAM interessa solitamente robot o veicoli autonomi. La sua risoluzione permette la creazione di una mappa a partire da un territorio sconosciuto, di cui non si ha alcuna conoscenza a priori, o l'aggiornamento di una mappa all'interno di un ambiente conosciuto. La mappa ottenuta può essere rappresentata come semplice schema bidimensionale, oppure presentarsi come una ricostruzione 3D del territorio esplorato. Il dispositivo si muoverà solitamente all'interno della zona circostante, raccogliendo informazioni che lo aiutino nell'atto di creazione/aggiornamento della mappa. Allo stesso tempo, il dispositivo dovrà tenere traccia della posa all'interno della stessa mappa, mantenendola coerente con le zone precedentemente mappate durante l'esplorazione.

Le due operazioni presentate risultano strettamente dipendenti fra di loro. SLAM infatti necessita di due cose: una mappa è richiesta per la localizzazione del dispositivo, mentre una stima della sua posa è richiesta per la creazione della mappa. Inoltre, la presenza di possibili errori di misurazione rende complicato l'intero procedimento. In particolare, tali errori possono propagarsi durante l'esecuzione di SLAM, causando risultati anche notevolmente imprecisi se la zona coperta va espandendosi considerevolmente.

Ad oggi la ricerca sul problema del Simultaneous Localization And Mapping si è frammentata, costruendo diverse strategie, ognuna delle quali segue un procedimento spesso anche molto diverso da quello delle altre.
Si descriverà nel seguito l'hardware tipicamente utilizzato per la risoluzione del problema SLAM, per poi soffermarsi sui principali approcci utilizzati per la realizzazione di un tale sistema, sia in termini di sensori utilizzati che in termini algoritmici.

1.2 Hardware utilizzato

L'hardware utilizzato allo scopo di permettere l'esecuzione di SLAM può variare considerevolmente in base al metodo utilizzato. Qui di seguito verranno dettagliate due tipologie di componenti fondamentali all'interno del sistema, evitando di concentrarsi sui dettagli implementativi.

- *Dispositivi di misurazione (sensori)*: Questi componenti svolgono il compito fondale di recupero delle informazioni dall'ambiente circostante. Il lavoro di tesi si concentrerà prevalentemente su informazioni di tipo visivo, benchè esistano una grande varietà di dati recuperabili mediante diversa strumentazione. Queste informazioni rappresentano dati fondamentali per la costruzione della mappa, nonchè per la localizzazione del dispositivo mobile. Esistono svariate tipologie di dispositivi che è possibile utilizzare a tale scopo, da semplici videocamere a più complicati sensori 3D o scanner laser. Non vi sono restrizioni sul tipo di dispositivo da utilizzare, a patto che le informazioni da esso generate siano sufficienti per il calcolo della posa e della mappa.
- *Dispositivo di calcolo*: Per l'esecuzione di SLAM è necessaria la presenza di un dispositivo dotato di capacità computazionali in quantità tale da permettere l'esecuzione dell'algoritmo. L'obiettivo ideale da raggiungere, in questo caso, riguarda la possibilità di avere un numero di risorse computazionali sufficienti ad elaborare i dati in real-time, essendo perciò in grado di sostenere la frequenza di

acquisizione tipica dei sensori utilizzati. Questo risultato non è di semplice realizzazione e non tutti gli algoritmi disponibili sono in grado di rispettarlo.

La piattaforma di calcolo è solitamente un robot o un veicolo autonomo con capacità computazionali simile a quelle di un classico compute desktop e adatte anche alla mappatura di territori inesplorati. Tuttavia, SLAM può essere utilizzato al semplice scopo di ricostruzione 3D di alcune zone, requisito che può essere facilmente soddisfatto anche con dispositivi più comuni, come ad esempio un tablet oppure uno smartphone, dove i requisiti in termini di memoria e tempo di esecuzione sono più stringenti e difficili da affrontare. Nessuna restrizione viene perciò definita riguardo alla tipologia di dispositivo utilizzato.

Oltre a questa distinzione, va notato che solitamente in un sistema SLAM i dispositivi sono in movimento, permettendo così la mappatura della zona. Tuttavia, non è strettamente necessario che l'intero sistema sia in movimento poichè anche solo lo spostamento dei sensori causerebbe la variazione delle informazioni in input al sistema. Solitamente si predilige un sistema compatto in cui dispositivi di misurazione e calcolo sono collegati fra di loro, ma questa non è l'unica soluzione.

1.3 Tipologie di SLAM in base al sensore

Data la notevole varietà di sensori attualmente disponibili, ognuno dotato di pregi e difetti, gli approcci di risoluzione di SLAM si sono diversificati notevolmente anche in questo ambito.

L'obiettivo di questo paragrafo è quello di indicare pregi e delle principali metodologie (dipendenti dal sensore) che sono state utilizzate nel corso degli anni.

1.3.1 SLAM con scanner laser

Un approccio spesso utilizzato per la risoluzione di SLAM prevede l'uso di uno scanner laser, il quale permette di acquisire dati sull'ambiente circostante in modo spesso molto preciso. Per ulteriori dettagli implementativi in merito a soluzioni precise, si rimanda a [MT04] e [BEetal08].

Il principale vantaggio nell'utilizzo di un sensore di questo tipo riguarda l'accuratezza delle misurazioni, spesso non raggiungibile con altre metodologie. Gli scanner laser risultano inoltre robusti rispetto all'ambiente che si sta mappando, in quanto sono insensibili a cambi di illuminazione ed ulteriori caratteristiche che altri sensori avrebbero maggiori difficioltà ad affrontare.

Gli svantaggi, però, sono svariati, in quanto gli scanner laser si presentano costosi e di dimensioni spesso elevate. Inoltre gli algoritmi che fanno uso di simili sensori si presentano computazionalmente complessi, limitando di fatto il campo di utilizzo di tale tecnologia. Per questi motivi la ricerca si è spinta verso soluzioni maggiormente sostenibili economicamente, nonchè di più pratico utilizzo e che richiedano un numero di risorse computazionali inferiore rispetto alla soluzione con scanner laser.

1.3.2 SLAM con dati RGB

Una soluzione che si differenzia molto dalla precedente prende il nome di Monocular SLAM (per implementazioni specifiche si rimanda a [DRetal07], [KM07] e [NLetal11]). Essa prevede l'utilizzo di dati RGB, ovvero immagini, tipicamente ottenute attraverso l'ausilio di una videocamera. Questi dati vengono utilizzati per stimare la posizione del dispositivo mobile e costruire una mappa del territorio, solitamente tramite l'ausilio di alcuni punti chiave all'interno dell'interno dell'immagine, chiamati *landmarks*.

Non avendo a disposizione dati di profondità per individuare la distanza dei singoli oggetti presenti nell'immagine, ci si appoggia a tecniche in grado di

stimare tali valori. La precisione potrà risultare inferiore rispetto al caso degli scanner laser, pur mantenendo un vantaggio in termini di costo e dimensioni dell'apparato utilizzato durante l'esecuzione dell'algoritmo.
Questo tipo di approccio ha già raggiunto una considerevole maturità, ma non abbastanza da permettere una ricostruzione 3D densa in tempo reale senza alcun tipo di accelerazione hardware, ad esempio tramite l'ausilio di algoritmi sviluppati per GPGPU. Inoltre è maggiormente sensibile a situazioni di illuminazione differente e non risulta utilizzabile in ambienti scarsamente illuminati o privi di texture.

1.3.3 SLAM con dati RGB-D

I difetti riscontrati nella soluzione precedente ci portano verso una terza soluzione, la quale ovvia ad alcuni dei problemi riscontrati dalla presenza dei soli dati RGB, pur mantenendo un costo non eccessivo.
La soluzione considerata prevede l'uso di dati RGB-D, in grado di rappresentare le informazioni con una immagine a colori ed una ulteriore struttura dati in grado di rappresentare la profondità dei singoli pixel presenti nell'immagine, intesa come la distanza fra il sensore e la posizione considerata. Dati di questo tipo possono essere raccolti tramite l'ausilio di sensori 3D, come ad esempio il sensore Asus Xtion Pro Live o il Microsoft Kinect, entrambi di dimensioni piuttosto contenute e perciò di più pratico utilizzo. Queste tipologie di sensori, di fatto, utilizzano una classica telecamera RGB per la ripresa delle immagini ed un sensore ad infrarossi per la misurazione della profondità. Questa impostazione già mette in luce il problema del sincronismo hardware dei due sistemi di cattura, non sempre reso disponibile dalla casa costruttrice e dunque da simulare via software.
Tramite l'ausilio di tali sensori si può raggiungere un buon livello di precisione, seppure non pari a quello degli scanner laser. Rimane d'altronde la difficoltà nel far fronte a repentini cambi di illuminazione, sebbene in tal caso si abbiano a disposizione anche i dati relativi alla profondità, che

potrebbero essere utilizzati per limitare tale problema. Inoltre, la presenza di sensori ad infrarossi rende difficile l'utilizzo di tali dispositivi in ambienti esterni, essendo facilmente confusi dalla luce solare.
Questa soluzione si presenta particolarmente adatta nel caso di ricostruzione 3D di ambienti illuminati artificialmente. È sicuramente meno adatta in caso di ambienti esterni o la cui illuminazione è notevolmente variabile o assente, anche a causa della loro limitata precisione rispetto agli scanner laser. Questo campo di ricerca risulta comunque piuttosto promettente, soprattutto per i costi ridotti uniti ad una buona qualità, sebbene limitata ad alcuni ambienti.
Un esempio di approccio di questo tipo è dato dall'algoritmo RGB-D SLAM illustrato in [EHetal12], il quale verrà in seguito analizzato.

1.4 Tipologie di SLAM in base all'algoritmo

Data l'estrema varietà di tipologie di sistemi SLAM, in questo paragrafo si descriveranno brevemente le principali, sottolineandone le differenze.

1.4.1 EKF SLAM

La prima strategia sviluppata per la risoluzione del problema SLAM ha visto l'utilizzo dell'Extended Kalman Filter (ulteriori dettagli in [Link01]).
Tale procedura si basa sull'estrazione di alcuni *landmarks* dall'ambiente circostante. Essi rappresentano punti salienti che il dispositivo è in grado di riconoscere facilmente tramite l'ausilio dei propri sensori e che sono utilizzabili dallo stesso per orientarsi all'interno dell'ambiente. Questi landmarks spesso dipendono dall'ambiente in cui il dispositivo si trova ad operare, nonchè dal metodo utilizzato per la loro individuazione, che può essere più o meno efficace e robusto alle diverse tipologie di trasformazione possibili.
L'Extended Kalman Filter, utilizzato dall'algoritmo, è un filtro ricorsivo in grado di valutare lo stato di un sistema dinamico a partire da una serie di misurazioni soggette a rumore (nel caso di SLAM le misurazioni sono

rappresentate dai dati provenienti dai sensori). Il filtro di Kalman classico si riferisce a sistemi lineari, mentre quello esteso, preso in considerazione nella risoluzione di SLAM, si riferisce a sistemi non lineari. Tale assunzione è veritiera in questo caso poichè il sistema in esame (ed il suo stato) evolve appunto in modo non lineare, in cui anche gli errori di osservazione sono in grado di influire non linearmente. Inoltre, i valori delle misurazioni considerate vengono definiti come appartenenti ad una distribuzione Gaussiana.

I passi compiuti dall'algoritmo sono i seguenti:

- Predizione dello stato in cui il sistema si troverà prossimamente;
- Predizione delle misurazioni effettuate, prevedendo cioè la posizione dei landmarks presenti nelle vicinanze;
- Misurazione reale della posizione dei landmarks;
- Associazione dei dati fra misurazione predetta e misurazione reale;
- Aggiornamento del sistema al nuovo stato.

La Figura 1.1 nel seguito mostra i passaggi compiuti da un punto di vista grafico, per meglio chiarire il funzionamento dell'algoritmo. Lo stato del sistema è rappresentato in termini matriciali. La matrice varia di dimensione in base al numero di landmarks considerati, aumentando quindi in complessità se il numero di landmarks è elevato ed impiegando di conseguenza più tempo per l'aggiornamento del sistema.

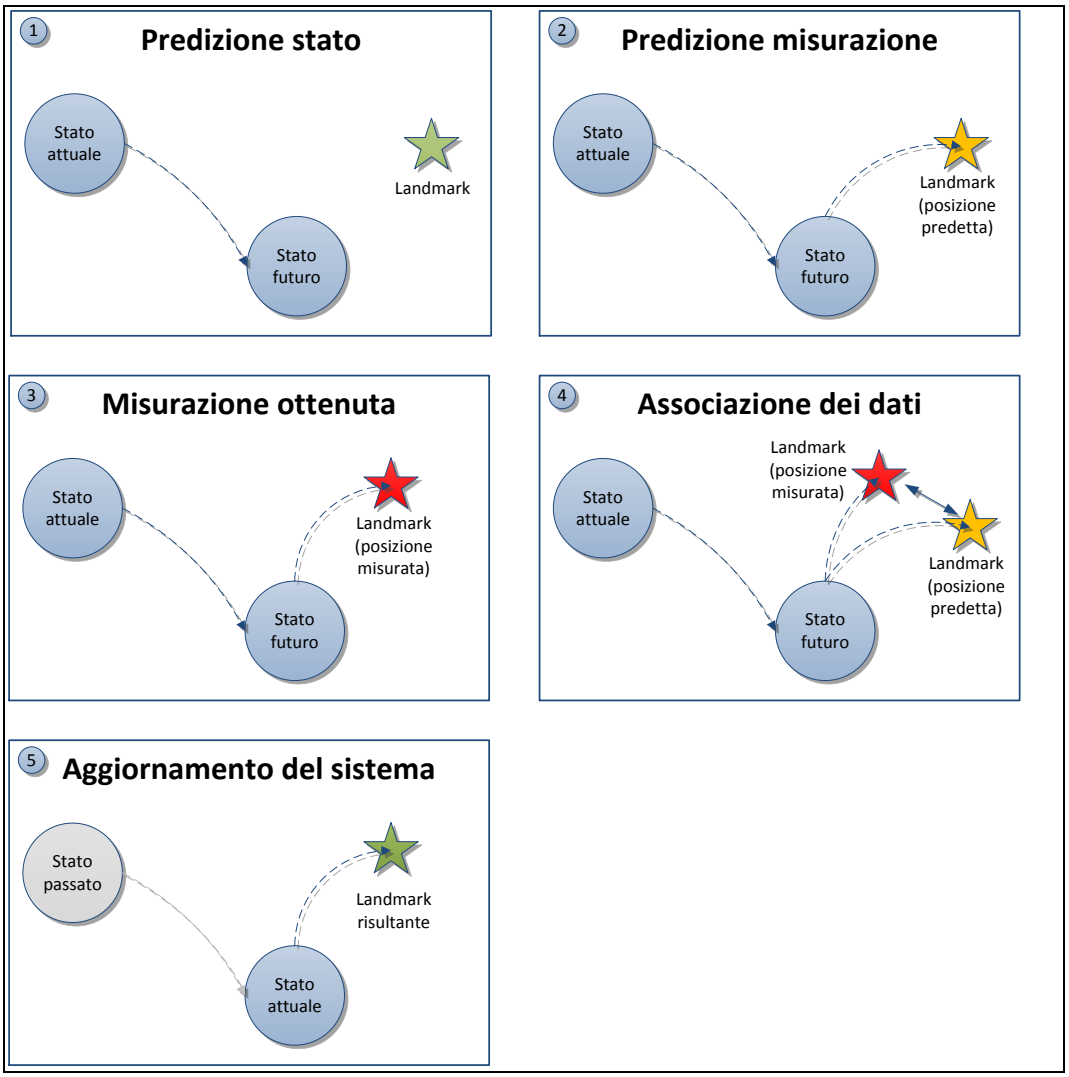

Figura 1.1:Passaggi dell'algoritmo EKF SLAM

Il principale lato negativo di questo primo approccio alla risoluzione di SLAM riguarda la modellazione di variabili in termini di distribuzioni gaussiane, cosa non sempre veritiera. Inoltre il sistema diventa computazionalmente dispendioso in termini di risorse nel caso di mappe piuttosto ampie, rendendone difficile l'esecuzione. Un altro punto negativo è la mancanza di una traiettoria continuamente aggiornata e migliorata, siccome le pose calcolate vengono dimenticate dall'algoritmo.

1.4.2 Graph-Based SLAM

Un successivo approccio considerato segue un'idea fondamentalmente diversa da quella precedente. Esso si basa su un grafo per effettuare

misurazioni e predizioni sul sistema, venendo così chiamato *Graph-Based SLAM* (ulteriori dettagli in [Link02] ed [Link03]). Questo approccio può essere inteso come una famiglia di algoritmi, le cui specifiche implementazioni variano in base ai casi considerati. Le considerazioni fatte in seguito si riferiscono ai casi più comuni di utilizzo del grafo.

Il grafo può essere così strutturato per rappresentare il problema:

- Ogni nodo nel grafo corrisponde ad una delle pose assunte dal dispositivo durante l'esecuzione dell'algoritmo;
- Ogni arco che collega due nodi rappresenta un vincolo fra le due pose.

L'algoritmo prevede l'utilizzo di un grafo allo scopo di rappresentare in modo semplice la funzione di costo associata allo stesso. In seguito alla costruzione dello stesso si cerca la giusta configurazione di nodi in grado di minimizzare gli errori introdotti dai vincoli, minimizzando di conseguenza la funzione di costo. La differenza rispetto ad EKF SLAM riguarda il fatto che, adottando una procedura di questo tipo, si prende in considerazione l'intera traiettoria attraversata dal dispositivo.

La gestione del grafo è solitamente delegata a due entità dagli scopi ben precisi:

- *Front-end*: Delegato alla costruzione stessa del grafo. Ottiene le informazioni di misurazione da parte dei sensori e le utilizza per individuare vincoli e creare nuovi nodi all'interno del grafo, spesso associandole ad informazioni ottenute precedentemente;
- *Back-end*: Si occupa dell'ottimizzazione del grafo risultante, minimizzando gli errori introdotti dai nuovi vincoli e dai nuovi nodi che il front-end inserisce durante l'esecuzione. L'ottimizzazione può essere svolta attraverso diversi metodi, di diversa precisione e complessità.

È importante sottolineare come non sia consigliabile l'ottimizzazione dell'intero grafo ogni volta che un nuovo vincolo è stato introdotto, a causa dell'elevato costo computazionale che l'operazione comporterebbe.

L'ottimizzazione parziale è in grado di velocizzare l'esecuzione dell'algoritmo rispetto ad una ottimizzazione totale svolta in seguito ad ogni aggiunta di vincoli, presentandosi come un giusto compromesso fra la qualità dei risultati e le prestazioni raggiunte.

Va inoltre notato come, in presenza di un approccio di questo tipo, sia relativamente semplice individuare situazioni in cui il dispositivo si ritrova in una locazione già precedentemente visitata (un caso definito di *loop closure*), avendo tenuto traccia nel grafo delle posizioni precedenti.

La successiva Figura 1.3 mostra un esempio di grafo con una possibile funzione di costo associata. Tale funzione è così definita (per maggiori informazioni riferirsi a [KGetal11]):

$$F(x) = \sum_{(i,j)} e(x_i, x_j, z_{ij})^T * \Omega_{ij} * e(x_i, x_j, z_{ij})$$

Da tale equazione Ω_{ij} rappresenta una matrice dei vincoli relativi ai parametri x_i ed x_j, mentre $e(x_i, x_j, z_{ij})$ è il vettore risultante dal calcolo di una funzione di errore, la quale misura il grado di percisione con cui i parametri x_i ed x_j soddisfano il vincolo z_{ij}. Il vettore vale 0 qualora il vincolo sia soddisfatto senza alcuna imperfezione. In figura questo vettore è espresso come e_{ij}.

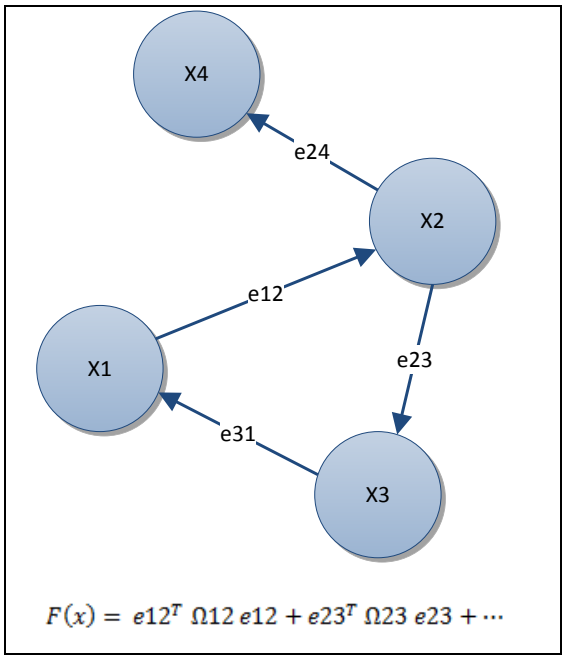

Figura 1.3: Grafo SLAM e funzione di costo

Gli algoritmi elencati di seguito prendono spunto dall'approccio a grafo per fornire una loro soluzione specifica al problema SLAM, modificando a loro vantaggio la natura del grafo precedentemente analizzato.

Adaptive Relative Bundle Adjustment:
L'algoritmo qui presentato [SMetal09] si basa sul problema del *Bundle Adjustment*, il quale si pone l'obiettivo di minimizzare gli errori presenti fra le misurazioni effettuate e predette di *n* landmarks ottenuti dalle *m* pose incognite del sensore. Un problema di questo tipo è solitamente risolto mediante una procedura di ottimizzazione non lineare ai minimi quadrati. Il costo di tale procedura è però cubico in complessità (in *n* o *m*), comportanto un tempo di esecuzione eccessivo con una quantità elevata di dati, ovvero per problemi ampi o che crescono rapidamente. Una condizione di particolare criticità riguarda i casi di loop closure precedentemente definiti: in queste situazioni è frequente la necessità di correggere tutti i parametri costituenti il loop, ma siccome gli errori si fanno più cosistenti mano a mano che il dispositivo si allontana dal punto di partenza, si potrebbe giungere alla condizione di loop con una quantità di errori tale da rendere infattibile il calcolo dell'ottimizzazione in un tempo costante.
Per ovviare a questi problemi l'algoritmo analizzato costruisce un grafo i cui nodi rappresentano dalle pose del dispositivo, specificando i landmarks collegati a ciascuna posa ed i vincoli associati. Il grafo si espande gradualmente a mano a mano che il dispositivo si muove nell'ambiente circostante, aggiungendo nodi ed archi. Grazie ad alcune considerazioni topologiche circa la natura del grafo, l'algoritmo è in grado di adottare strumenti matematici che cosentano una ottimizzazione efficiente ed adattativa, riferita soltanto ad un sottoinsieme dei parametri necessari al calcolo ed escludendo quelli che meno influiscono sull'andamento dei risultati. Viene adottata una struttura dati chiamata *active* window, la quale ha il compito di definire l'area entro cui verrà effettuata l'ottimizzazione del grafo. In questo modo l'algoritmo è in grado di eseguire in tempo di

esecuzione costante, presentandosi maggiormente robusto all'aumentare della quantità di dati e garantendo buone performance nei casi di loop closure.

Double Window Optimisation:

Questo secondo algoritmo analizzato [SDetal11] si riferisce al ramo di SLAM relativo alle informazioni visive, ovvero Visual SLAM. Esso prende in considerazione il lavoro svolto nell'implementazione dell'Adaptive Relative Bundle Adjustment, il quale non effettua mai operazioni di ricostruzione globale, mantenendo invece procedure locali per far fronte ai problemi di performance precedentemente riscontrati. Un difetto dell'approccio è dato però dal fatto che esso non assicura consistenza metrica all'interno della finestra di ottimizzazione, rendendo possibile un progressivo degrato dell'accuratezza nel caso di loop ripetuti all'interno di un'area ristretta.

Per ovviare a tale problema si decide di applicare l'approccio ad active window con la differenza che in questo caso le finestre sono due. Una *inner window* memorizza i vincoli tipici del problema di Bundle Adjustment, ovvero fra landmarks e pose, in modo similare all'algoritmo precedente, mentre una *outer window* rappresenta i vincoli fra le pose. La inner window ha il compito di gestire in modo preciso l'ottimizzazione nell'area locale, mentre la outer window stabilizzerà la ricostruzione all'esterno di tale area.

Un approccio di questo tipo migliora di fatto le performance dall'algoritmo precedente e ne corregge i casi più critici, riducendo gli errori presenti in tali situazioni.

RGB-D SLAM:

L'ultimo approccio che si analizzerà [EHetal12] è riferito esclusivamente a sensori RGB-D e verrà descritto nei suoi passaggi principali, anzichè concentrarsi esclusivamente sulla rappresentazione del grafo come nei casi precedenti.

La procedura adottata consiste nell'analisi dei frame RGB e depth in ingresso all'algoritmo, individuando nell'immagine RGB alcuni landmarks utili ad un suo riconoscimento. In seguito a questa operazione, vengono cercate le corrispondenze fra questi punti e i landmarks memorizzati da immagini precedenti (operazione di *feature matching*). Viene selezionato solo un numero ristretto di immagini su cui cercare le corrispondenze, per ovviare a problemi di performance. Le corrispondenze fra i punti vengono poi proiettate nello spazio 3D, grazie alla presenza delle informazioni di profondità, e ulteriormente filtrate facendo uso dell'algoritmo *RANSAC* [FB81]. Dalle corrispondenze risultanti si può ottenere la posa corrispondente al frame correntemente analizzato. Essa viene inserita in un grafo, aggiungendo i relativi vincoli con le pose già presenti all'interno dello stesso e collegate a quest'ultima. Si procede infine all'ottimizzazione complessiva del grafo. L'output dell'algoritmo viene visualizzato attraverso alcune point cloud.

La Figura 1.4 mostra i passaggi dell'algoritmo, che si presenta come un ottimo punto di partenza per l'analisi del successivo approccio alla risoluzione di SLAM, sviluppato nell'Università di Bologna ed in parte simile all'algoritmo appena descritto. Le differenze con esso si presentano però di notevole importanza, tanto da consentire un aumento elevato delle performance, che nel caso di RGB-D SLAM erano piuttosto ridotte, principalmente a causa dell'ottimizzazione globale del grafo.

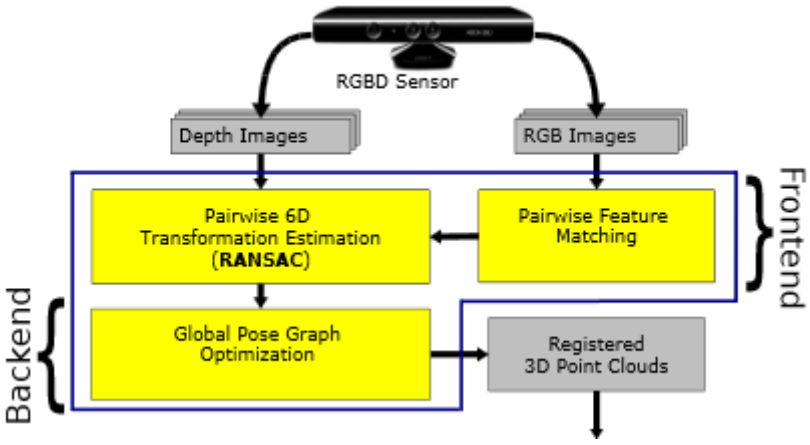

Figura 1.4: Passaggi fondamentali effettuati da RGB-D SLAM

Capitolo 2
SlamDunk

Nel capitolo precedente sono stati descritti scopi e funzioni di un sistema SLAM, nonchè presentato i principali approcci e sensori adottati. Assieme a ciò sono state sottolineate le diverse problematiche legate ai singoli algoritmi ed ai sensori utilizzati, individuando così le soluzioni migliori in base alle specifiche esigenze da soddisfare.

In questo capitolo verrà descritto un algoritmo, chiamato *SlamDunk* [FD13], il quale utilizza dati provenienti da sensori RGB-D e mappa le pose calcolate all'interno di un grafo. L'algoritmo analizzato è in grado di operare in real-time su di un personal computer, senza richiedere alcuna accelerazione hardware. Per tale motivo si apre la possibilità di realizzarne una implementazione su dispositivi mobili, rendendo fattibile una ricostruzione 3D degli ambienti anche con hardware comunemente disponibile sul mercato.

2.1 Pipeline dell'algoritmo

Nel seguito verrà descritta la pipeline dell'algoritmo, indicando i principali moduli di cui esso è composto ed i passaggi effettuati dallo stesso, a partire dai frame RGB-D ricevuti in input, fino a giungere alla posa definita in output ad ogni iterazione, assieme ad una indicazione sull'andamento dell'algoritmo (fallimento del tracking, frame tracked o keyframe tracked). Nei prossimi paragrafi verranno inoltre ulteriormente dettagliati i principali moduli del sistema, per meglio comprenderne lo scopo ed il funzionamento.
L'algoritmo utilizzato può essere inizialmente decomposto in 3 moduli distinti, qui descritti. Il primo modulo è chiamato *Camera Tracking* e riceve i dati RGB-D provenienti dal sensore in uso. Lo scopo di questo modulo è la stima della posa relativa al frame in input. A questo fine vengono inizialmente individuate delle *features* all'interno dell'immagine RGB, utilizzando diversi possibili algoritmi di computer vision (tipicamente SIFT [L04] o SURF [BTetal06] per le loro proprietà di invarianza e ripetibilità). Le features identificano dei punti chiave all'interno dell'immagine, i quali hanno proprietà utili e spesso indipendenti dall'orientamento o dalla scala dell'immagine. Tali features vengono in seguito comparate, tramite operazione di matching, con altre ottenute da frame precedenti (memorizzate all'interno di una struttura dati indicizzata chiamata *Feature Pool*).
Va qui notato come non vi sia la necessità di effettuare un matching sull'intero catalogo dei frames precedentemente collezionati, innanzitutto poichè molti di essi rappresentano pose che nulla hanno a che fare con quella che si vuole calcolare in questa iterazione. Inoltre, non tutti i frame precedenti verranno memorizzati dall'algoritmo; soltanto un sottoinsieme degli stessi, che prendono il nome di *keyframes*, verrà mantenuto. Il matching avverrà considerando i keyframes nelle vicinanze della posizione corrente della camera.
In seguito al matching, i risultati vengono filtrati facendo uso dell'algoritmo iterativo *RANSAC* [FB81], eliminando possibili match non corretti.

Successivamente al filtering, la posa del dispositivo viene stimata basandosi sui match rimanenti.

Il secondo modulo, chiamato *Local Mapping*, modella il percorso effettuato dal dispositivo. In seguito all'operazione di tracking, si calcola la percentuale di overlapping del frame correntemente esaminato rispetto ai keyframes considerati per il matching. Se tale percentuale è inferiore di una determinata soglia, si è in presenza di un nuovo keyframe che dovrà essere aggiunto agli altri. I keyframes raccolti, assieme alle pose relative, vengono mantenuti all'interno di un *quadtree*, ovvero una struttura dati ad albero utilizzata comunemente per partizionare uno spazio bidimensionale. In questo caso tale struttura dati viene utilizzata per poter efficientemente individuare keyframes presenti all'interno di una *active keyframe window*, centrata sulla posizione corrente, allo scopo di selezionare soltanto i keyframes nelle vicinanze della posizione corrente della camera. La active keyframe window si muoverà inoltre assieme al dispositivo, mantenendo la ricerca coerente con la posizione dello stesso.

L'ultimo modulo componente il sistema è chiamato *Local Optimization* ed effettua una ottimizzazione delle pose ottenute, per minimizzare gli errori. Questa operazione è similare a quanto precedentemente descritto per gli algoritmi SLAM basati sui grafi, ricordando che i vincoli in questo caso sono vincoli fra pose.

Le pose dei keyframes, oltre ad essere memorizzate all'interno del quadtree precedentemente definito, costituiscono anche i nodi di un grafo. All'aggiunta di ogni posa diversi vincoli si aggiungono all'interno del grafo ed una ottimizzazione è richiesta. Tale ottimizzazione, per motivi di efficienza, viene svolta localmente, rendendo così maggiormente scalabile l'algoritmo. Va infine sottolineato come, grazie alla presenza della mappa locale fornita attraverso la active keyframe window, sia possibile individuare facilmente casi di loop closure, avviando una ottimizzazione locale anche in queste situazioni.

La Figura 2.1 nel seguito mostra complessivamente la pipeline dell'algoritmo, con tutti i passaggi principali all'interno dei relativi moduli.

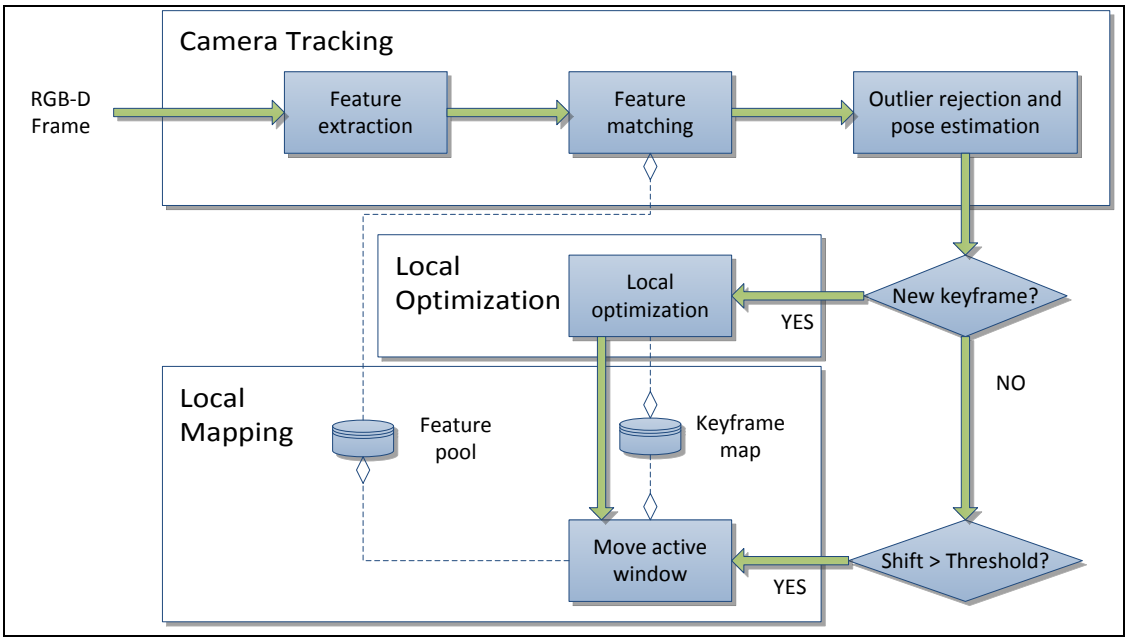

Figura 2.1: SlamDunk pipeline

Verranno di seguito analizzati i dettagli implementativi dei diversi moduli considerati, in modo tale da comprendere meglio il funzionamento del sistema.

2.2 Camera Tracking

Lo stadio iniziale della procedura di stima della posa corrente della camera è costituito dall'estrazione delle features nel frame RGB-D in ingresso. Tale operazione prevede innanzitutto la detection dei *keypoints* componenti l'immagine RGB, ovvero dei punti caratteristici all'interno della stessa, utili per distingere lo specifico frame dagli altri. Basandosi sull'intorno di tali keypoints, un elenco di descrittori, o *feature descriptors*, è estratto dall'immagine. Esistono svariati algoritmi di Computer Vision in grado di estrarre features e/o individuare keypoints, uno dei più utilizzati è sicuramente SURF, il quale è in grado di ottenere una accuratezza elevata in un tempo di calcolo accettabile in ambiente desktop. Una pecca di tale algoritmo riguarda però la sua impraticabilità nell'adottarlo sui moderni dispositivi mobili, ancora troppo limitati dal punto di vista hardware per poter eseguire l'algoritmo in tempi accettabili. Anche in questi casi però

esistono algoritmi molto più rapidi, sebbene più imprecisi, come ad esempio ORB [RRetal11], che verrà successivamente analizzato.

In seguito all'estrazione delle features, viene eseguita l'operazione di matching con le features presenti nel pool. Tale operazione utilizza delle tecniche avanzate di indicizzazione allo scopo di cercare, per ogni descrittore dell'immagine, i K descrittori più vicini nello spazio ad alta dimensionalità delle features. La ricerca appena descritta viene anche chiamata *k-nearest neighbor search*. Non ci si accontenta, perciò, di individuare la feature più vicina a quella scelta, bensì se ne selezionano di più. Lo scopo di tale operazione è quello di comparare due features appartenenti allo stesso keyframe e che più si avvicinano a quella scelta, verificando che il rapporto delle loro distanze dalla feature in esame sia inferiore ad una certa soglia (il *ratio test* proposto da Lowe [L04]), ovvero che la seconda feature non si trovi in prossimità della prima. Data la presenza di diversi keyframes, è possibile che la prima feature appartenga ad un determinato keyframe mentre la successiva ad un keyframe diverso. In questo caso si manterrà la prima feature e si cercherà la successiva all'interno dello stesso frame, che non è detto sia la più vicina in termini assoluti, bensì esclusivamente all'interno del keyframe in considerazione. La ragione di questo test è data dal fatto che molto spesso le features non corrette si trovano ad una distanza simile fra di loro, rispetto alla feature su cui è stato effettuato il matching. Introducendo questa operazione si è in grado di filtrare un numero piuttosto elevato di match scorretti, mantenendo quelli più veritieri. Nel caso si utilizzi l'algoritmo SURF per l'estrazione delle features, il matching verrà solitamente attuato tramite l'ausilio di un KD-Tree per effettuare la ricerca, con possibilità di essere esplorato in parallelo.

Come ulteriore filtraggio dei match trovati, siccome spesso possono esserne rimasti alcuni non corretti, viene effettuata una operazione di *outlier rejection* utilizzando l'algoritmo iterativo RANSAC. Nello specifico, per ogni coppia di features trovata, i pixel associati vengono inizialmente proiettati nello spazio 3D. Per effettuare tale operazione bisogna

innanzitutto tenere presente i parametri intrinseci del sensore preso in considerazione, indicati tramite la matrice *K* così costituita:

$$K = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}$$

I parametri f_x ed f_y indicano la focale nelle direzioni della camera considerata, mentre c_x e c_y indicano il centro della visuale della stessa. Tali parametri, inseriti all'interno della matrice, sono fondamentali per poter proiettare un punto 2D dell'immagine RGB, indicato dalle coordinate (*u*, *v*), nello spazio 3D, facendo uso inoltre del parametro relativo alla profondità, indicato come *D* e sempre riferito al pixel tenuto in considerazione. Il punto 3D ricavato, indicato come *p'*, è così definito:

$$p' = K^{-1} * \begin{pmatrix} u * D \\ v * D \\ D \end{pmatrix}$$

L'algoritmo prevede la proiezione, ad ogni iterazione, di 3 coppie di punti, scelte casualmente. Successivamente alla proiezione si effettuano calcoli per trovare la trasformazione, in termini di rotazione e traslazione, in grado di portare i 3 punti dell'immagine corrente a combaciare con i 3 punti di cui fanno match (procedura dettagliata in [AHetal87]). Ottenuta la trasformazione, si può applicare la stessa a tutti i punti del set, contando quanti di essi distano meno di un certo valore soglia rispetto ai match con cui fanno coppia. Il valore ottenuto da tale conteggio, rappresentante il numero di match corretti, è utile per stimare la probabilità di estrarre un inlier per questo modello, ovvero un punto interno allo stesso, rappresentata da $w = \frac{numero\ di\ match\ corretti}{numero\ totale\ di\ match}$. Detti *n* i punti necessari alla stima di un nuovo modello, con *n* uguale a 3 in questo caso, ne segue che w^n sia una ragionevole stima della probabilità di estrarre nuovamente *n* punti appartenenti al modello e di conseguenza $(1 - w^n)$ rappresenta il caso opposto, ovvero la probabilità che almeno uno di questi sia un outlier e, come tale, dia luogo ad un modello diverso. Ripetendo il processo per *k* iterazioni, la probabilità di non estrarre mai n-ple di punti inlier al modello è data da: $(1 - w^n)^k$. Assumento di conoscere a priori la probabilità *p* di

estrarre, in una certa iterazione, *n* punti inlier al modello reale che si sta cercando di stimare, si ottiene la seguente equazione:

$$1 - p = (1 - w^n) * k$$

Prendendo in considerazione il logaritmo, si ottiene infine un buona stima del numero di iterazioni necessario per accettare come valida una determinata stima della trasformazione:

$$k = \frac{\log(1 - p)}{\log(1 - w^n)}$$

Le operazioni di trasformazione vengono ripetute fino a quando non si supera il numero massimo di iterazioni. Una volta ottenuto il risultato finale, i restanti punti potranno essere scartati e si procederà ad una stima ai minimi quadrati della trasformazione tenendo in considerazione soltanto i punti rimanenti.

La posa trovata, al tempo *i*, viene definita tramite la seguente matrice:

$$T_i = \begin{pmatrix} R_i & t_i \\ \overline{0} & 1 \end{pmatrix}$$

dove R_i è una matrice di rotazione 3x3 e t_i è un vettore di traslazione nelle tre coordinate cartesiane. A partire dalla posa così indicata, il passaggio da pixel dell'immagine a punti 3D in coordinate globali avviene mediante riproiezione, come precedentemente discusso, e successiva roto-traslazione come indicato dall'operazione seguente:

$$p_i = R_i * [p_i'] + t_i$$

Una volta tracciato il movimento della camera, si giunge alla decisione sulla promozione a keyframe del frame corrente. A questo scopo, il frame attualmente in esame viene diviso in un certo numero di celle e vengono contate le celle nelle quali cadono più di F features per cui è stato trovato un match corretto. Se tale numero è inferiore ad una certa soglia, il frame corrente viene considerato come keyframe e perciò mantenuto in memoria assieme alla propria posa. Questo approccio si presenta molto semplice, eppure notevolmente efficace in svariati casi, poichè assume che le features presenti all'interno del frame seguano una distribuzione uniforme.

2.3 Local Mapping

L'insieme dei keyframe raccolti al muoversi del dispositivo nell'ambiente è memorizzato ed indicizzato, secondo la posizione della camera nello spazio tridimensionale, in una struttura dati nota come quadtree. Essa si basa sulla suddivisione ricorsiva dello spazio nel piano parallelo alla direzione principale del moto della camera. La suddivisione avviene facendo uso di una struttura ad albero e dividendo lo spazio in 4 nodi rappresentanti 4 spazi distinti, i quali a loro volta possono essere suddivisi allo stesso modo fino a quando una determinata condizione non è soddisfatta, ad esempio il raggiungimento del livello massimo dell'albero, generando così i nodi radice. La suddivisione non viene effettuata all'atto di creazione dell'albero, ma soltanto quando in una determinata posizione viene inserito un keyframe. Come si può notare, soltanto due delle 3 coordinate spaziali sono indicizzate mentre l'altezza è scartata, supponendo una traiettoria che mantenga all'incirca lo stesso posizionamento verticale.

Per recuperare i keyframes dal quadtree così creato, è sufficiente eseguire una operazione di query specificando la posizione centrale, ovvero l'ultima posizione nota della camera, e le dimensioni della finestra considerata, ovvero la active window necessaria per il tracciamento del dispositivo. L'operazione di query è effettuata ogni qualvolta l'algoritmo individua un nuovo keyframe o, in generale, quando è richiesto l'aggiornamento del Feature Pool (vedi Figura 2.1).

La scelta di utilizzare una struttura dati di tipo quadtree semplifica la realizzazione e migliora le performance dell'algoritmo, rispetto a soluzioni più complesse come ad esempio un octree, il quale terrebbe in considerazione anche l'altezza come coordinata spaziale. Il lato negativo riguarda appunto un vincolo aggiuntivo all'applicazione dell'algoritmo, in quanto si suppone che la camera venga mossa principalmente lungo la direzione orizzontale, parallelamente al terreno considerato. Se così non fosse, una struttura dati di tipo quadtree potrebbe non ottimizzare le

prestazioni, siccome potrebbero essere presi in considerazione anche keyframes molto distanti fra di loro.

Una ulteriore struttura dati è utilizzata per la memorizzazione delle features, la quale prende il nome di Feature Pool. Questa struttura dati è implementata come KD-Tree, una rappresentazione ad albero simile al quadtree ma più generale in quanto prevede una dimensionalità più elevata. Questa struttura dati rappresenta un albero binario in cui ogni nodo può essere visto come una suddivisione dello spazio in due parti, lungo una delle direzioni spaziali possibili, fino al raggiungimento dei nodi radice dell'albero. L'utilizzo del KD-Tree ottimizza la ricerca concentrandosi su zone specifiche dello spazio e tralasciando perciò le features presenti in altre locazioni. Inoltre si considera il frustum della posa attuale, il quale è in grado di indicare la posizione osservata dal sensore nell'iterazione corrente, per scartare eventuali features che si trovino al di fuori dello stesso.

Infine, esiste la necessità di poter muovere l'active window anche in assenza dell'identificazione di un nuovo keyframe, questo poichè, se il dispositivo ritorna in zone già precedentemente mappate, potrebbe non esserci la necessità di creare nuovi keyframes, pur essendo però necessario l'aggiornamento della finestra. Se tale operazione venisse tralasciata, il calcolo dell'overlapping individuerebbe nuovi keyframes anche qualora non ve ne sia la necessità, riducendo perciò le performance dell'applicazione e aggiungendo dati non utili. Per ovviare a tale problematica, l'active window viene spostata dall'algoritmo non appena il dispositivo ha percorso una certa distanza (indicata come Shift nella Figura 2.1) superiore ad una determinata soglia, evitando così un aggiornamento ad ogni nuovo frame ricevuto.

La Figura 2.2 nel seguito mostra graficamente il quadtree con all'interno la active window, considerando la posa centrale (indicata con il cerchio) ed alcune pose vicine (i triangoli), alcune delle quali fuori dalla finestra considerata. La presenza di spazi vuoti è giustificata dal fatto che in tali posizioni non sono stati aggiunti elementi, e non si è resa dunque necessaria

una suddivisione. Nel caso della figura il quadtree si ferma una volta raggiunto il terzo livello.

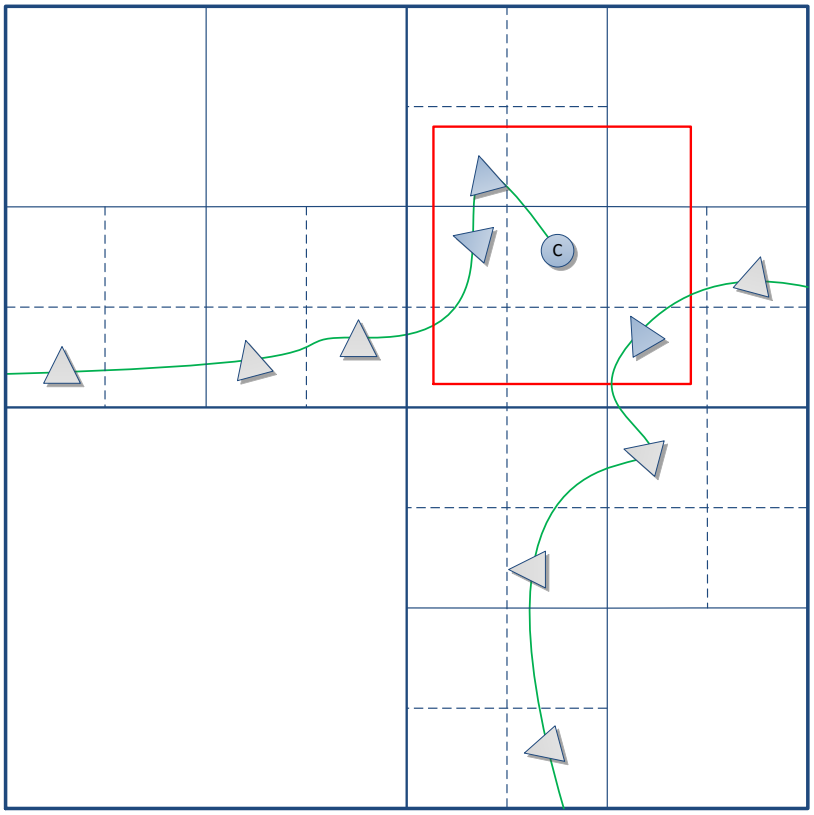

Figura 2.2: Pose all'interno del quadtree

2.4 Local Optimization

L'individuazione di un nuovo keyframe porta una notevole quantità di nuove informazioni all'interno del sistema, alle volte individuando anche casi di loop closure. In queste situazioni risulta necessaria un'operazione di ottimizzazione che possa influire anche sulle pose precedentemente memorizzate, minimizzando gli errori introdotti nel sistema. Questa operazione è fondamentale per ottenere buoni risultati nelle differenti esecuzioni dell'algoritmo, migliorando la qualità della ricostruzione ottenuta.

L'algoritmo prende in considerazione un grafo in cui ogni nodo rappresenta una posa assunta dal dispositivo, ed ogni arco un vincolo fra due pose,

come già descritto nella Sezione 1.4.2 del Capitolo 1. Si vuole perciò minimizzare la seguente funzione di costo:

$$\mathrm{F}(T_0, \dots, T_{n-1}) = \sum_{(m_i, m_j) \in \varphi} w_{ij} \left\| e_{ij} \right\|^2$$

In tale espressione, { T_0, ..., T_{n-1} } sono le pose sconosciute di cui si intende stimare il valore, l'inseme φ rappresenta i matches fra i keyframes considerati e $w_{ij} \left\| e_{ij} \right\|^2$ è il costo associato al match fra il keyframe *i* ed il keyframe *j*. Specificando ulteriormente l'ultima parte dell'espressione, il peso w_{ij} sta ad indicare il grado di confidenza del match: è un valore compreso fra 0 ed 1, maggiore è il valore, migliore è il match in termini di accuratezza. Tale valore viene calcolato basandosi sul rapporto delle distanze precedentemente calcolato durante il ratio test (vedi la Sezione 2.2 del'attuale Capitolo): il valore risultante dal rapporto viene sottratto ad 1, ottenendo il peso indicato. Per quanto concerne la seconda parte dell'espressione, si tiene in considerazione un vincolo fra i punti proiettati nello spazio 3D: il termine e_{ij} può essere scritto come la differenza fra i punti 3D associati ai matches, una volta espressi entrambi i punti nel medesimo sistema di riferimento. L'equazione che descrive tale procedimento è indicata di seguito:

$$e_{ij} = p_i' - T_i^{-1} * T_j * \left[p_j' \right]$$

Il calcolo descritto non verrà effettuato globalmente, ovvero considerando tutti i keyframes memorizzati, bensì si ottimizzano esclusivamente i nodi nelle vicinanze di quello che attualmente rappresenta l'ultimo keyframe individuato. La ricerca nell'intorno avviene tenendo in considerazione l'ultimo nodo come radice del grafo e procedendo con una ricerca breadth-first: i nodi raggiunti da tale ricerca, all'interno dei primi R livelli, verranno considerati nel problema di ottimizzazione. Solitamente si sceglie un valore di R pari a 3, che sta ad indicare i nodi ad una distanza massima di 3 dal nodo radice scelto. Va sottolineato inoltre come gli ultimi nodi trovati nella ricerca (ovvero quelli appartenenti al livello R) non partecipino

direttamente nell'ottimizzazione, bensì soltanto nella determinazione dei vincoli da considerare all'interno del problema.

La Figura 2.3 mostra i nodi considerati nel caso di un numero di livelli (anche chiamati *anelli*) pari a 3. Il nodo R rappresenta la radice del grafo, i nodi X1 ed X2 rappresentano i nodi interni trovati nel livello 1 e 2 nella ricerca, i nodi B rappresentano i nodi fissi presenti al livello 3 della ricerca, mentre i nodi indicati con O rappresentano i nodi esclusi dall'ottimizzazione locale.

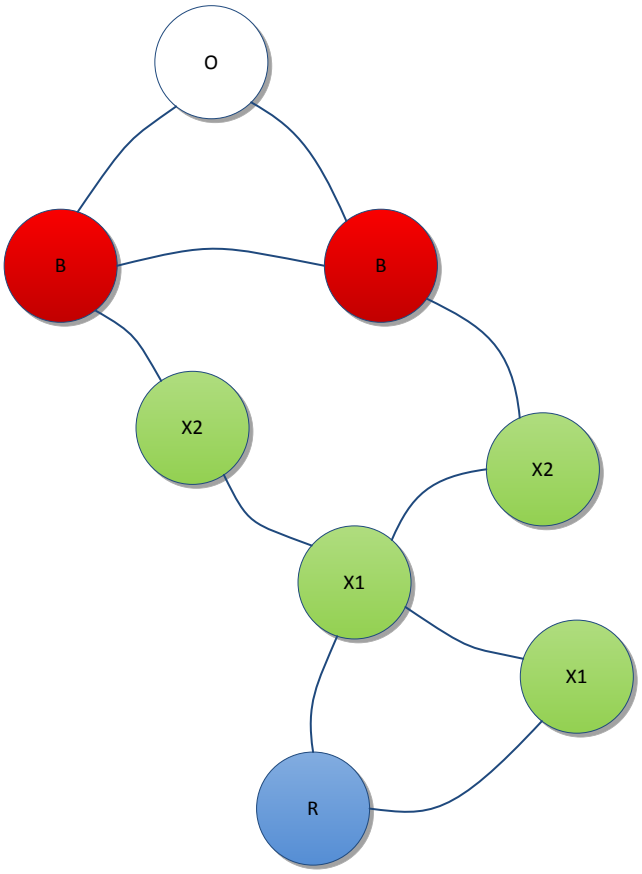

Figura 2.3: Rappresentazione dei nodi da ottimizzare

Il procedimento utilizzato permette di ridurre il tempo impiegato nell'ottimizzazione pur ottenendo buoni risultati in termini di precisione. Esso si presenta inoltre molto semplice da implementare ma è comunque una semplificazione del caso generale che comporterà una minore precisione rispetto al caso di ottimizzazione globale.

Infine, per quanto concerne il caso di loop closure, l'algoritmo già indirettamente ne tiene in considerazione grazie all'adozione della active window, la quale è in grado di prelevare pose fra di loro anche distanti temporalmente ma vicine in termini spaziali. In questo caso, però, l'ottimizzazione verrebbe attuata esclusivamente quando un nuovo keyframe è stato trovato. Per far sì che essa possa essere adottata anche nel caso singolo di loop closure, l'algoritmo implementa un ulteriore controllo sul grafo, in grado di individuare la situazione di loop anche in assenza del ritrovamento di un nuovo keyframe. Ciò permette una maggiore precisione a discapito di un tempo di calcolo maggiore, avendo abilitato l'ottimizzazione anche in casi diversi dall'inserimento di un nuovo keyframe.

2.5 Differenze rispetto ad altri algoritmi

Accanto a SlamDunk, in letteratura è possibile trovare altre proposte di risoluzione del problema SLAM basate esclusivamente su dati RGB-D. In questa sezione prenderemo in considerazione il sistema RGB-D SLAM [EHetal12] precedentemente descritto e particolarmente simile a SlamDunk, cercando di evidenziare le differenze fra i due.

Nel sistema RGB-D SLAM le features vengono estratte e confrontate per trovare corrispondenze fra il frame attuale ed alcuni di quelli precedenti. In seguito viene effettuato un filtraggio tramite RANSAC. La differenza principale in questa parte dell'algoritmo riguarda il fatto che in SlamDunk il matching delle features avviene non solo rispetto ai frame precedenti, bensì rispetto ad ulteriori frames, prelevati attraverso la struttura dati quadtree e la active window precedentemente descritta. Questi frames non sono collegati fra di loro in termini temporali, come invece avviene nel caso di RGB-D SLAM, bensì in termini spaziali. In tal modo si ottiene una maggiore precisione rispetto all'algoritmo considerato.

Una ulteriore differenza riguarda la gestione del caso di loop closure. RGB-D SLAM effettua dei controlli separati per verificare tale situazione,

eseguendo un matching con un sottoinsieme dei frame precedentemente analizzati. Nel caso di SlamDunk, invece, il caso di loop closure viene gestito durante i movimenti del sensore e non in modo separato dalle restanti parti del sistema, consentendo quindi una maggiore coerenza nell'implementazione.

Infine, una delle differenze più importanti riguarda il tempo di esecuzione degli algoritmi, in quanto RGB-D SLAM viene eseguito su un pc desktop ad una velocità di 2/3 Hz, circa 10 volte più lentamente di SlamDunk. È proprio questa velocità che spinge l'algoritmo verso una possibile implementazione su piattaforma mobile, la quale potrebbe raggiungere velocità accettabili con buoni risultati in termini di ricostruzione 3D. Ciò si analizzerà a partire dal prossimo capitolo, in cui verranno messe in luce le particolari esigenze da tenere in considerazione per il porting di SlamDunk su dispositivo mobile.

Capitolo 3
Analisi dello sviluppo su piattaforma mobile

Una volta introdotto l'algoritmo SlamDunk ed aver individuato i suoi punti di forza rispetto ad algoritmi similari, è ora necessario analizzare dettagliatamente le implicazioni collegate ad un porting dell'algoritmo su un dispositivo mobile. Queste considerazioni sono motivate dal fatto che tali dispositivi differiscono notevolmente dai computer desktop, essendo nati in seguito agli stessi ed essendo ancora agli albori del loro sviluppo tecnologico.

Una prima evidente differenza riguarda i metodi di sviluppo su tali dispositivi, che spesso appaiono semplificati per poter così facilitare la realizzazione di semplici applicazioni di interazione con l'utente, ma diventano vincolanti qualora l'applicazione da realizzare fosse dotata di una maggiore complessità. Strettamente legato a questa prima differenza, vi è il problema della compatibilità con i diversi programmi per computer desktop, spesso non facilmente eseguibili in un ambiente mobile. Ciò si rispecchia anche nell'incompatibilità con alcune periferiche esterne, essendo i driver delle stesse tipicamente non progettati per un utilizzo su dispositivi mobili. Questo aspetto è particolarmente decisivo, in quanto il dispositivo scelto dovrà interfacciarsi ad un sensore 3D, risultando così indispensabile la compatibilità dello stesso con il dispositivo mobile.

Alle precedenti considerazioni si aggiunge la necessità di ottimizzare le risorse in ambiente mobile, sfruttando le potenzialità a disposizione della piattaforma Android scelta per questo progetto e dell'hardware considerato. Ciò permetterà di velocizzare l'esecuzione dell'algoritmo sulla piattaforma, il quale altrimenti rischierebbe di essere troppo lento per un utilizzo relativamente fluido.

Per ognuna delle considerazioni affrontate si indicheranno le possibili soluzioni, evidenziando quelle che verranno implementate e le motivazioni alla base di tali scelte.

La Figura 3.1 nel seguito fornisce uno schema generale dei diversi componenti che verranno analizzati in questo capitolo. Essi costituiscono la struttura dell'intera applicazione desktop su cui si intende effettuare il porting, non considerando solo l'algoritmo in sè ma anche le modalità di interfacciamento con i sensori disponibili e con l'interfaccia grafica.

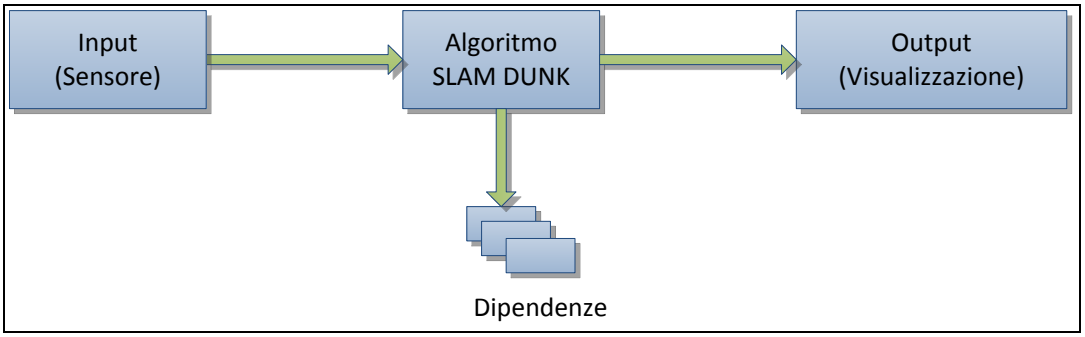

Figura 3.1: Componenti del sistema da analizzare

3.1 Analisi delle dipendenze dell'algoritmo

In questo primo paragrafo verranno considerate le dipendenze che l'algoritmo SlamDunk ha rispetto ad alcune librerie software esterne, individuando se tali dipendenze possano risultare un problema all'atto del porting o se si possono risolvere in un qualche modo.

Innanzitutto va notato come l'implementazione desktop di SlamDunk sia attualmente stata realizzata tramite codice C++. Questa prima considerazione non risulta comunque problematica in quanto, sebbene il linguaggio di programmazione ufficiale delle applicazioni Android sia Java, Google mette a disposizione un Native Development Kit (NDK), tramite il quale è possibile programmare utilizzando codice C/C++, richiamato attraverso opportune API Java. Ulteriori informazioni riguardo all'NDK ed ai metodi di programmazione verranno illustrati nei prossimi capitoli, maggiormente incentrati sulla parte pratica della soluzione, mentre per ora

viene data una maggiore importanza alla semplice analisi delle diverse problematiche cui far fronte.

Oltre all'utilizzo di C++ come linguaggio di programmazione, diverse librerie esterne sono state utilizzate in appoggio alla realizzazione dell'algoritmo. Tali librerie verranno descritte qui di seguito, indicando per ognuna le caratteristiche ed i motivi per cui è stata utilizzata, nonchè le possibilità o meno di adoperarla in ambiente Android. Essendo ancora in una fase di analisi, eventuali dettagli implementativi verranno tralasciati per essere esposti in una fase successiva del seguente elaborato.

3.1.1 OpenCV

OpenCV [G00] rappresenta sicuramente la più popolare libreria open-source di Computer Vision, la quale ospita le implementazioni dei maggiori algoritmi per la keypoint detection, l'estrazione delle features ed il loro matching, operazioni fondamentali per SlamDunk. Tale libreria offre ulteriori funzionalità, come ad esempio la memorizzazione delle immagini in apposite strutture dati ottimizzate per una loro rapida manipolazione (tipi di dato Mat), con la possibilità di convertire le stesse in diversi formati. Una conversione particolamente utile in questo caso è quella da immagini RGB a grayscale, essendo che la keypoint detection viene solitamente effettuata su immagini grayscale. È anche presente un modulo di I/O in grado di leggere o scrivere le immagini, semplificando così ulteriori metodologie di testing dell'applicazione. Inoltre, essendo distribuita con licenza open-source, questa la libreria ha l'ulteriore vantaggio di essere liberamente modificabile in base alle proprie necessità. Ciò potrà essere sfruttato per adattare gli algoritmi utilizzati a poter meglio adoperare le risorse disponibili sulla specifica piattaforma mobile in uso.

Un porting Angroid di OpenCV è già disponibile e facilmente utilizzabile sia tramite interfaccia Java, sia da codice nativo, senza richiedere complesse configurazioni

Tuttavia alcune delle funzionalità presenti nella versione desktop non sono ufficialmente disponibili su Android, tra cui gli algoritmi SIFT e SURF, quest'ultimo utilizzato nella versione desktop di SlamDunk. Il motivo di ciò è dovuto al fatto che tali algoritmi sono protetti da brevetto o licenze restrittive: un loro utilizzo in determinate applicazioni, prevalentemente commerciali, è consentito soltanto previo pagamento di royalties. In ogni caso, per scopi di ricerca, tali algoritmi possono essere utilizzati liberamente e, nel nostro caso specifico, è stato possibile ricompilare la libreria introducendo in essa il modulo che li contiene. La loro adozione non è generalmente conveniente su piattaforma mobile, a causa della complessità degli algoritmi, i quali potrebbero richiedere un tempo di esecuzione elevato. Tramite ottimizzazioni hardware si presenta però la possibilità di utilizzo degli stessi, seppur presentando una maggior complessita di realizzazione.

3.1.2 FLANN

L'implementazione desktop di SlamDunk utilizza una particolare libreria per effettuare il matching delle features, chiamata *FLANN* (Fast Library for Approximate Nearest Neighbors [ML09]). Questa libreria fornisce efficienti algoritmi di indicizzazione per le ricerche di elementi vicini (nearest neighbor search), permettendo anche una esecuzione parallela basata sul numero di core del processore attualmente in uso. Nel contesto del feature matching è possibile sfruttare FLANN sia in presenza di descrittori binari, ovvero stringhe di bit in grado di rappresentare direttamente alcune misurazioni, che in presenza di descrittori fromati da vettori di numeri in virgola mobile, i quali rappresentano solitamente alcuni istogrammi. Ad esempio, in SIFT la rappresentazione è data da istogrammi delle direzioni dei gradienti presenti in una patch centrata sul keypoint in considerazione. In ORB, invece, la rappresentazione è di tipo binario, occupando un numero fisso di bit. Ad ogni tipo di feature corrisponde dunque un tipo di matching diverso, che sfrutta le particolarità della specifica rappresentazione.

L'implementazione della libreria FLANN è in codice C/C++ ed un suo porting in ambiente Android richiede solamente una compilazione della libreria in modo simile a quanto potrebbe avvenire nel classico caso desktop. Va però notato come gli stessi algoritmi implementati nella libreria FLANN siano già disponibili all'interno di OpenCV, rendendo di fatto non necessario un suo utilizzo diretto. Essendo ambo le librerie open-source, anche in questo caso non vi sono problemi in caso si volessero aggiungere ottimizzazioni al codice. Inoltre, parte della libreria è già ottimizzata per un matching efficace in ambiente Android, utilizzando per tale scopo istruzioni in codice assembler.

3.1.3 Eigen

Dovendo affrontare diverse operazioni matriciali durante l'esecuzione dell'algoritmo, specialmente durante la fase di rimozione degli outlier, risulta utile sfruttare una libreria esterna che fornisca tali operazioni e sia ottimizzata per limitare il tempo di esecuzione ed il numero di risorse occupate. Per tale motivo l'implementazione di SlamDunk include la libreria *Eigen* [GB10] per effettuare la maggior parte delle operazioni matematiche, come ad esempio moltiplicazioni fra matrici o il calcolo dell'inversa. Tale libreria è scritta anch'essa in C++ ma, al contrario delle altre già analizzate, ha la particolarità di essere una libreria realizzata esclusivamente tramite header files, non rendendo perciò necessaria una sua compilazione.

La libreria non presenta particolari problemi nell'essere utilizzata in ambiente Android e risulta anche ottimizzata per tale sistema. Un lato negativo è però rappresentato dal fatto che l'utilizzo della libreria allo stato attuale richiede l'invocazione dei metodi da codice nativo C++, operazione che non è sempre consigliata a causa dell'overhead introdotto, specialmente in caso di operazioni semplici che potrebbero essere mantenute in Java senza particolari problemi. Per tale motivo si è ritenuto utile modificare la realizzazione di SlamDunk per far sì che su sistema Android utilizzi Eigen

soltanto nelle operazioni più complesse computazionalmente, mantenendo così un buon livello di performance e delegando i calcoli più semplici a delle classi Java appositamente realizzate per questo scopo. Ciò sicuramente aggiunge del lavoro da svolgere all'interno del porting ma permettere di ottenere una realizzazione più pulita e facile da modificare in futuro. Inoltre, la presenza delle classi Java può essere di utilità anche in altri moduli presenti esclusivamente nell'applicazione Android, basti pensare ad esempio alla parte relativa all'interfaccia grafica, in cui alcune operazioni matriciali sono sicuramente necessarie. La realizzazione Java di buona parte delle operazioni svolte su desktop tramite Eigen permette di trasformare gran parte del codice C++ di SlamDunk in codice Java, risultando però in un calo visibile delle performance che comprometterebbe l'esecuzione dell'algoritmo. Per tale motivo si affiancherà alla versione Java una versione interamente C++ dell'algoritmo, utilizzata nei casi in cui è richiesto il massimo vantaggio in termini di performance. Essendo tale versione di più difficile modificabilità e meno supportata nell'ambiente Android, in un futuro, aiutati dallo sviluppo in ambito tecnologico che comporterà prestazioni sempre più elevate, si mirerà al mantenimento della sola versione Java.

3.1.4 G2O

L'algoritmo analizzato deve far fronte al problema dell'ottimizzazione di un grafo, il quale potrebbe essere implementato utilizzando diverse metodologie che introducono però una maggiore complessità realizzativa per l'applicazione generale. Per far fronte in modo semplice a tale problematica si utilizza il framework *g2o* [KGetal11], appositamente sviluppato per la risoluzione di problemi di ottimizzazione ai minimi quadrati con rappresentazione sparsa su grafo. Questo framework presenta alcune caratteristiche tipiche dei sistemi SLAM, essendo esso stesso realizzato anche allo scopo di facilitare l'implementazione di tali sistemi, a partire dalla possibilità di rappresentazione di un grafo e dalla specifica di

vincoli negli archi con valori personalizzabili in base all'algoritmo adottato (nel caso di SlamDunk i vincoli sono espressi come indicato nella Sezione 2.4 del Capitolo 2). È inoltre dotato di diverse metodologie di ottimizzazione e risoluzione, ad esempio l'utilizzo di solver basati sulla decomposizione di Cholesky o solver PCG (preconditioned conjugate gradient), permettendo così una scelta in grado di adattarsi agli specifici requisiti di ogni singola applicazione.

G2O è scritto in C++ e può essere compilato in Android, avendo come unico requisito la disponibilità di Eigen che, come già visto precedentemente, non presenta problemi su tale piattaforma. Come quest'ultimo, però, nell'utilizzo della libreria ci si dovrà obbligatoriamente basare su codice C++, scelta non ottimale dal punto di vista della riusabilità in ambiente Android, la quale permette però un miglioramento considerevole delle performance. In tal caso la complessità risulta maggiore nel caso si volesse realizzare tale libreria in Java, anche parzialmente, per cui si è deciso di mantenere la libreria compilata con codice C++, anche in virtù del fatto che il suo utilizzo è presente soltanto nella fase di ottimizzazione, ovvero principalmente quando un nuovo keyframe si aggiunge nel sistema, evento non particolarmente frequente.

3.1.5 Boost

Infine, SlamDunk utilizza anche una popolare libreria C++, chiamata *Boost*, per la gestione della memoria e dei processi all'interno del codice. Tale libreria risulta di fondamentale importanza nel caso di applicazioni C++ ma un suo uso in ambiente Android, dove la Virtual Machine è già in grado di gestire tali problematiche, risulta non necessario.

Nella versione del porting che preve la sostituzione di buona parte del codice C++ con codice Java, tale libreria risulterebbe inutilizzata e perciò non ne risulta necessaria l'introduzione.

Nella versione che manterrà buona parte del codice di SlamDunk in C++, tale libreria dovrà essere compilata per ambiente Android per essere così

utilizzata all'interno del codice. Fortunatamente è possibile compilare la libreria senza incorrere in problemi, grazie specialmente ad alcuni sviluppatori indipendenti che hanno lavorato per rendere tale libreria disponibile anche su dispositivi mobili.

3.2 Gestione dell'input

Fino ad ora sono state considerate soltanto le particolarità riferite ai moduli interni dell'algoritmo, ovvero tutto ciò che viene eseguito una volta ottenuto il frame RGB-D in input, sino all'ottenimento della posa in output.

A questo punto risulta necessario considerare in che modo il frame RGB-D viene originato e come esso viene poi passato in input all'algoritmo. Questa fase è spesso dipendente dal tipo di sensore utilizzato, necessitando dei driver dello stesso per consentire al sistema operativo di interfacciarsi con l'hardware del sensore. Per tale motivo, l'operazione risulta in qualche modo scomoda nel caso sia necessario cambiare sensore, dovendo riprogrammare parte della logica applicativa realizzata per l'acquisizione dei frame RGB-D.

Tutto ciò viene parzialmente facilitato dalla presenza del framework *OpenNI* [O11], il quale fornisce API condivise e funzionanti in modo simile per diversi sensori, come ad esempio per il Microsoft Kinect e per l'Asus Xtion Pro Live, attualmente a disposizione dell'Università di Bologna. L'unica operazione necessaria per il cambio di sensore riguarda la compilazione dei driver per l'hardware apposito, i quali si interfacceranno alle API OpenNI comuni ai diversi dispositivi. Sono inoltre presenti diverse impostazioni che permettono, ad esempio, di cambiare la risoluzione del frame RGB o depth, nonchè di abilitare o meno le diverse caratteristiche peculari del dispositivo connesso.

L'applicazione desktop di SlamDunk fa attualmente uso di OpenNI attraverso un'altra libreria, chiamata Point Cloud Library [RC11]. Essendo tale libreria molto più complessa e dotata di moduli che non sono di interesse nella versione Android, si è deciso di optare invece per un utilizzo

diretto di OpenNI. Ciò è possibile grazie anche al fatto che OpenNI, assieme alle dipendenze dello stesso, è compilabile su ambiente Android ed utilizzabile attraverso codice C/C++ con relativa facilità. Si è scelto di utilizzare la versione 1.5 del framework, in quanto compatibile con entrambi i sensori sopra descritti, mentre nella versione attuale (versione 2) il Kinect non è fino ad ora supportato.

Per quanto riguarda considerazioni di natura hardware, va ricordato che non tutti i dispositivi mobili potranno interfacciarsi con dei sensori 3D di questo tipo. Innanzitutto la gran parte dei dispositivi mobili è dotata di una singola porta Micro-USB, mentre i sensori sono dotati di collegamento USB, per cui è necessario un cavo che permetta una conversione fra questi due formati, chiamato cavo OTG. Oltre a questa considerazione, è necessario che il dispositivo mobile sia dotato della funzionalità di USB Host, senza la quale il dispositivo potrebbe non essere in grado di comunicare in modo appropriato con il sensore.

Oltre ai due sopra citati esistono altri sensori RGB-D sul mercato, i quali andranno via via esaminati in base alla possibilità di funzionamento su piattaforma Android. Va ricordato che, essendo il sistema Android basato su Linux, molti driver che supportano questo sistema potrebbero essere facilmente compilati per essere compatibili con la piattaforma Android. Ciò aumenta le possibilità che i driver per futuri sensori 3D vengano resi disponibili anche in ambiente mobile. Inoltre, alcuni sensori sono stati sviluppati appositamente per l'ambiente mobile e stanno ora prendendo piede (ad esempio il sensore Structure [Link04]), a dimostrazione del fatto che questo campo mostra buone prospettive di crescita.

Come ultima considerazione, una volta ottenuti i dati che il sensore ha emesso, la loro manipolazione dovrebbe risultare particolarmente facile, rendendoli così disponibili all'algoritmo SlamDunk in un formato base unificato e perciò indipendente dal sensore utilizzato. Ciò è dato dalla natura stessa dei dati considerati e spesso manipolati attraverso array di bytes per quanto riguarda l'immagine RGB o di floating points per la depth map.

Per i motivi sopra elencati la gestione dell'input su dispositivo Android non dovrebbe risultare problematica da realizzare, sebbene sia necessario compiere alcune operazioni di compilazione dei driver e di scrittura del codice per la lettura e la manipolazione dei frame RGB-D. Va infine ricordato che, per facilitare l'inserimento delle librerie OpenNI all'interno del dispositivo Android, potrebbe essere necessario ottenere i privilegi di root (ovvero amministratore del sistema) sul dispositivo stesso. Tale operazione non risulta comunque impegnativa e perciò è di facile raggiungimento, permettendo inoltre diverse libertà su altri vincoli che potrebbero essere presenti all'atto di implementazione di SLAM, pur ricordando che, nel caso si volesse realizzare una versione che non richiede privilegi speciali, bisognerà trovare una alternativa alla soluzione realizzata.

3.3 Gestione dell'output

Una ulteriore fase da considerare riguarda la gestione dell'output dell'algoritmo, ovvero in che modo vengono utilizzati i dati relativi alla posa fornita in output. Oltre a questo dato, l'algoritmo SlamDunk fornisce anche una indicazione su cosa è avvenuto nell'iterazione dell'algoritmo. I casi sono 3:

- *Fallimento nel tracking*: In questo caso il dato di posa in output non verrà considerato in quanto i dati del tracking non sono validi. Questo caso avviene solitamente qualora i matches ottenuti siano pochi oppure le features ritrovate non siano sufficienti.
- *Frame tracked*: È stato effettuato il tracking del dispositivo nel frame corrente, ovvero la posa è valida ma il frame considerato non è un keyframe. In questo caso solitamente non avviene alcuna visualizzazione di point cloud in output.
- *Keyframe tracked*: È stato riconosciuto un nuovo keyframe, ovvero la posa è valida ed il keyframe è stato aggiunto nella mappa dei keyframes. In questo caso in output possono essere richieste tutte le pose dei keyframes che sono state modificate in seguito

all'ottimizzazione, per poter modificare la loro posizione nella fase di visualizzazione. Oltre a ciò, una nuova point cloud riferita al nuovo keyframe verrà visualizzata.

Dalla definizione di questi 3 casi si sottolinea come in seguito all'esecuzione dell'algoritmo sia presente una fase di visualizzazione, la quale non viene tipicamente aggiornata ad ogni iterazione dell'algoritmo.

Questa fase prevede la visualizzazione di alcune point cloud riferite ai keyframes analizzati, attuata proiettando i pixel delle immagini RGB nello spazio 3D in modo similare a quanto visto nel capitolo precedente riguardo l'algoritmo RANSAC, includendo anche, oltre moltiplicazione di tali punti con la matrice *K* della camera, una moltiplicazione con la posa *T* appena calcolata. La visualizzazione delle sole point cloud riferite ai keyframes è motivata dal miglioramento di performance così ottenuto, nonchè dal fatto che l'aggiunta di una point cloud ad ogni iterazione rappresenterebbe uno spreco ingente di risorse ed una visualizzazione ridondante dei punti appartenenti alle point cloud.

Anche in questo caso, la versione desktop di SlamDunk si appoggia alla Point Cloud Library per quanto concerne la visualizzazione 3D delle point cloud, con annessa la possibilità di movimento nella scena così generata. Per quanto concerne la versione Android si è deciso di realizzare la visualizzazione con codice autonomo, essendo la stessa di semplice realizzazione ed offrendo possibilità di ottimizzazione in ambiente mobile. Android nello specifico si basa su OpenGL ES per quanto riguarda la visualizzazione 3D, permettendo l'implementazione della scena 3D con relativa facilità e consentendo una maggiore personalizzazione, non appoggiandosi ad alcuna libreria esterna. Un ulteriore pregio riguarda la possibilità di programmazione della parte di rendering in codice interamente Java, senza tuttavia comportare un calo evidente in termini di performance.

Ulteriori informazioni riguardanti l'implementazione di tale soluzione verranno date nel prossimo capitolo.

3.4 Analisi delle risorse su dispositivi mobili

Passiamo ora ad esaminare il problema della scarsità di risorse disponibili in ambiente mobile.

Come precedentemente discusso, la programmazione su dispositivi mobili deve spesso tenere in considerazione la relativa scarsità di risorse computazionali, specialmente per quanto riguarda l'utilizzo della CPU e l'occupazione della memoria centrale (RAM).

Per quanto concerne la CPU, i dispositivi mobili sono spesso dotati di processori ARM in grado di sostenere frequenze pari a 2 GHz e 4 core fisici. Tali processori, sebbene potenti e di anno in anno sempre più efficienti, non risultano paragonabili ai processori desktop e possono risultare anche 4 o 5 volte meno performanti, fino addirittura ad una decina di volte nei casi più estremi. Ciò mette in luce la necessità di una ottimizzazione più elevata in ambiente mobile, cosa che spesso non risultava necessaria su computer desktop, specialmente nei casi in cui l'algoritmo raggiungeva velocità real-time anche senza specifiche ottimizzazioni. Per poter raggiungere velocità quantomeno accettabili in ambiente mobile risulta dunque consigliabile una accelerazione hardware delle parti del codice più critiche, ad esempio mediante l'utilizzo di codice su GPU, qualora ciò fosse possibile, oppure adottando linguaggi di più basso livello per aumentare l'efficienza del codice, ad esempio utilizzando l'assembler ARM, specialmente le istruzioni dell'architettura ARM NEON attualmente disponibili in un numero elevato di dispositivi mobili.

Per quanto riguarda, invece, l'occupazione di RAM, va ricordato che la piattaforma Android limita in modo considerevole il quantitativo di RAM che ogni singola applicazione è libera di riservare. Tale quantitativo è decisamente inferiore alla dimensione reale della memoria volatile ed è stato aggiunto per evitare che alcune applicazioni occupassero un quantitativo di RAM talmente elevato da compromettere l'esecuzione delle altre applicazioni presenti all'interno del dispositivo. Questa limitazione può essere in parte ovviata attraverso alcune modifiche alla configurazione,

sebbene sia consigliabile una strategia che prevenga l'occupazione eccessiva della memoria, senza modificare configurazioni di sistema. A tale scopo sono possibili diverse strategie, specialmente mirate ad una riduzione dell'occupazione di memoria relativa alle point cloud visualizzate su schermo. La strategia più semplice è data da un subsampling di tali point cloud, ovvero una riduzione del numero di punti effettivamente contenuti nelle stesse. Altre tecniche da maggiore complessità potrebbero prevedere un salvataggio su disco di alcune point cloud, specialmente se esse non vengono visualizzate sullo schermo in quel momento, ed un caricamento delle stesse quando necessario. Tutte queste considerazioni potranno essere discusse con maggiore dettaglio in seguito, non costituendo comunque lo scopo principale dell'attuale lavoro di tesi, maggiormente incentrato sulle performance dell'algoritmo in riferimento al tempo di esecuzione ed ai risultati di precisione ottenuti.

Capitolo 4
Realizzazione del porting

Dopo aver discusso le caratteristiche da tenere in considerazione per effettuare il porting di SlamDunk su piattaforma mobile, si è ora giunti nella fase di progettazione ed implementazione stessa del porting.

Durante questa fase si è scelto di utilizzare un tablet Android come macchina di sviluppo, le cui specifiche verranno indicate in seguito, nel Capitolo 5 dedicato ai risultati sperimentali. Inoltre, è stato possibile lavorare con diversi sensori ed anche con immagini campione, descritte nel capitolo di testing poichè forniranno informazioni cruciali sulla precisione raggiunta dall'algoritmo.

Data la natura sperimentale dell'applicazione, sono state effettuate prove utilizzando diversi algoritmi, specialmente per quanto riguarda la keypoint detection e la feature extraction, nonchè il successivo matching. Lo scopo è stato quello di individuare la soluzione che potesse ottimizzare maggiormente il rapporto fra qualità e performance. Molto spesso infatti, soluzioni maggiormente precise (come ad esempio l'utilizzo dell'algoritmo SURF) sono realizzate a discapito del tempo di esecuzione, parametro che si rivela cruciale in ambito mobile e che costringe la ricerca di soluzioni alternative.

Lo scopo di questo capitolo è quello di mostrare le scelte progettuali adottate per lo sviluppo dell'applicazione, evitando tuttavia di addentrarsi troppo in alcuni dettagli implementativi, bensì preferendo un approccio di più alto livello, che possa essere in gran parte utile anche in porting studiati per altre piattaforme mobili o addirittura in altri progetti di ottimizzazione similari. Ovviamente risulta comunque necessario introdurre alcune particolarità della piattaforma Android, come ad esempio l'uso dell'NDK per l'utilizzo di codice C/C++.

4.1 Utilizzo dell'NDK Android

Prima di addentrarsi nella parte principale dello sviluppo del porting, risulta utile una digressione sull'utilizzo dell'NDK Android, unito all'SDK Java che viene utilizzato nelle classice applicazioni sviluppabili su questa piattaforma. Questa introduzione è necessaria a chiarire i diversi casi in cui l'NDK è stato utilizzato e le sue particolarità. Per ulteriori dettagli sullo sviluppo in ambiente Android si rimanda al sito ufficiale gestito da Google [Link05].

Per lo sviluppo dell'applicazione Android si è utilizzato *eclipse* come ambiente di sviluppo, su cui è possibile installare l'SDK Android per la programmazione Java e l'NDK Android per la programmazione in codice nativo C/C++. In tal modo sarà possibile programmare direttamente dall'IDE utilizzando sia codice Java che codice C/C++.

Lo sviluppo di codice nativo può essere abilitato o meno per ogni progetto eclipse. Il termine *nativo* deriva dal fatto che il codice C/C++ è in grado di essere compilato facilmente per l'hardware specifico del sistema. Se in un progetto è abilitato lo sviluppo nativo, è possibile inserire in una specifica directory i codici sorgente e gli header dell'applicazione C/C++, richiedendo la realizzazione di alcuni makefile (Android.mk ed Application.mk) molto simili ai classici makefile Linux, per la compilazione e l'utilizzo di tale codice. Lo scopo dei due makefile sopra indicati è illustrato nel seguito:

- *Application.mk*: Makefile contenente le impostazioni generiche riferite al codice C/C++, ad esempio la versione Android minima supportata, l'architettura su cui compilare i sorgenti (Intel x86, ARM, ARM v7, ...), i flag del compilatore C o C++ e la Standard Template Library (STL) da utilizzare. Solitamente è importante indicare l'architettura, nel caso si voglia supportare solo una di esse, e la versione minima di Android supportata. La STL è in grado di fornire una serie di classi comuni spesso fondamentale nello sviluppo C++, per cui anch'essa riveste un ruolo importante. In

questo caso come libreria STL è stata utilizzato il parametro gnustl_static, il quale sta ad indicare la libreria statica GNU;

- *Android.mk*: Questo makefile è più dettagliato del precedente e permette di inserire indicazioni sui path dei file sorgente e degli headers considerati all'interno dell'applicazione. Oltre a ciò è possibile indicare le librerie statiche e/o dinamiche a cui il codice è collegato ed il nome della libreria che verrà generata in seguito alla compilazione dei sorgenti. È possibile generare diverse librerie, ognuna riferita a diversi sorgenti e compilabile per diverse architetture. È importante notare come si possa effettuare il link sia di librerie attualmente presenti sulla macchina di sviluppo, le quali verranno perciò inserite all'interno dell'applicazione per essere utilizzabili dalla stessa all'atto di esecuzione, sia di librerie presenti all'interno del dispositivo mobile. In questo secondo caso l'ambiente di sviluppo supporrà la loro esistenza all'interno del dispositivo e ciò verrà verificato all'atto di esecuzione dell'applicazione. Questo file permette anche l'attivazione delle istruzioni assembler ARM NEON, nel caso in cui il sistema su cui il codice verrà eseguito supporti tali istruzioni.

In seguito viene mostrato un esempio relativo a questi due file, per meglio chiarire il loro contenuto.

Application.mk:

```
APP_STL := gnustl_static
APP_CPPFLAGS := -frtti -fexceptions
APP_ABI := armeabi-v7a
APP_PLATFORM := android-9
```

Android.mk:

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)

LOCAL_MODULE := LibraryName
LOCAL_SRC_FILES += path/file_source.cpp
LOCAL_ARM_NEON := true
LOCAL_LDLIBS += path/to/system/lib/file_shared.so
LOCAL_LDLIBS += path/to/system/lib/file_static.a
```

```
LOCAL_SHARED_LIBRARIES += path/to/local/lib/file_shared.so
LOCAL_STATIC_LIBRARIES += path/to/local/lib/file_ static.a
LOCAL_C_INCLUDES += path/to/includes

include $(BUILD_SHARED_LIBRARY)
```

Sulla parte relativa all'Application.mk non vi è altro da aggiungere rispetto a quanto precedentemente detto. Invece, per l'Android.mk, si possono notare due righe aggiuntive all'inizio del file: la prima serve a ottenere il path locale e salvarlo in una variabile, per possibili utilizzi futuri; la seconda serve invece a resettare il valore precedentemente assunto da alcune variabili locali, come ad esempio quelle indicanti i file sorgente o le librerie associate. In questo modo sarebbe possibile compilare diverse librerie utilizzando lo stesso makefile, avendo cura di resettare le variabili di interesse fra la compilazione di una libreria ed il setup della successiva. Come si può notare la compilazione avviene tramite il comando indicato nell'ultima riga dell'esempio, producendo in questo caso una libreria shared con nome LibraryName (il nome completo del file risulterà libLibraryName.so). Come ulteriore aggiunta, si sottolinea che l'utilizzo della variabile LOCAL_LDLIBS va effettuato nel caso di librerie di sistema (ovvero presenti all'interno del dispositivo fisico vero e proprio, nelle cartelle di sistema), mentre l'utilizzo delle variabili LOCAL_SHARED_LIBRARIES e LOCAL_STATIC_LIBRARIES si riferisce a librerie locali al computer su cui si sta sviluppando, le quali andranno perciò inserite nell'applicazione senza essere preventivamente presenti all'interno del dispositivo mobile.

Giunti a questo punto si è già in grado di comprendere come sia possibile compilare una generica applicazione C/C++ in ambiente Android, però deve essere ancora affrontata la modalità con cui si è in grado di interagire con tale codice. Per fare ciò Android sfrutta un framework inizialmente introdotto in ambiente Java, chiamato *Java Native Interface* (JNI).

Tramite questo framework è possibile dichiarare nel codice Java alcuni metodi con la keyword *native*, privi di implementazione.

Questi tipi di metodi verranno implementati in un file sorgente C/C++ utilizzando una speciale sintassi per rendere il link fra i due metodi univoco. La sintassi utilizzata è la seguente:

```
Java_native_package_NativeClass_nativeMethod(params)
```

Con essa è possibile specificare il package della classe Java associata, il suo nome ed il nome del metodo di cui effettuare il link, specificando in seguito gli ulteriori parametri dello stesso. Due parametri aggiuntivi presenti all'interno di ogni metodo JNI sono dati da un oggetto di classe JNIEnv e da un oggetto generico di classe jobject.

Nello specifico, JNIEnv è utile nella costruzione di array Java da codice C/C++, utilizzati ad esempio come valore di ritorno di un metodo JNI. Tramite questa classe è anche possibile ottenere array passati da codice Java a codice C/C++, ovvero presenti come parametro all'interno del metodo. Va notato che i tipi di dato utilizzati da JNI per indicare i tipi primitivi o gli array non sono gli stessi utilizzati da C, bensì dei wrapper con cui viene facilitata l'interazione fra i due linguaggi. Qualora si volessero utilizzare i dati primitivi C sarà dunque necessaria una conversione.

Riguardo l'oggetto di classe jobject, esso non contiene altro che un riferimento alla classe su cui è stato invocato questo metodo, cosa utile in determinate occasioni, ad esempio qualora si volessero invocare metodi della stessa classe cui appartiene il codice C/C++ qui generato.

Come ultimo accorgimento, prima di eseguire qualsiasi metodo nativo riferito ad una determinata libreria, è necessario caricare la stessa, specificandone il nome, attraverso l'invocazione di un metodo da codice Java, nel seguente modo:

```
System.loadLibrary("LibraryName");
```

Terminata questa introduzione all'NDK Android ed al framework JNI è utile elencare i pro e i contro di questo approccio.

Da un lato, si beneficia dall'utilizzo di tecnologie già ampiamente diffuse, il quale permette un possibile riutilizzo del codice realizzato in ambiente Java

desktop, nonchè una relativa facilità nello sviluppo, considerando l'ampia documentazione a disposizione riguardante JNI e la programmazione con tale framework. L'utilizzo di una nuova tecnologia avrebbe sicuramente comportato un maggiore investimento e scoraggiato gli sviluppatori, non propensi all'apprendimento di un nuovo sistema eclusivamente per poter interfacciare codice Java a C++.

Dall'altro lato, va notato che JNI non si presenta come un framework di facile utilizzo ed alcune migliorie nella fase di deployment, relative ad esempio al link fra metodo C/C++ e metodo Java, avrebbero potuto permettere la realizzazione di soluzioni più eleganti. Inoltre non è presente una trasparenza nell'utilizzo dei dati, essendo i tipi di dato diversi fra di loro e richiedendo quasi obbligatoriamente alcune operazioni di conversione. Infine, il passaggio di dati è fonte di overhead elevato, soprattutto qualora gli stessi siano numerosi. Ciò è inevitabile e comporterà la necessità di ridurre al minimo le chiamate a metodi nativi, o almeno ridurre la quantità di dati trasferita.

4.2 Struttura generale dell'applicazione

Una volta introdotta la parte maggiormente platform specific, ovvero l'NDK Android utilizzato nello sviluppo del codice C/C++, è ora possibile mostrare lo schema generale di funzionamento dell'applicazione. Questo schema, mostrato nella Figura 4.1, si presenta simile a quanto mostrato nel capitolo precedente, ma in questo caso le entità coinvolte rappresentano classi o interfacce reali dell'applicazione.

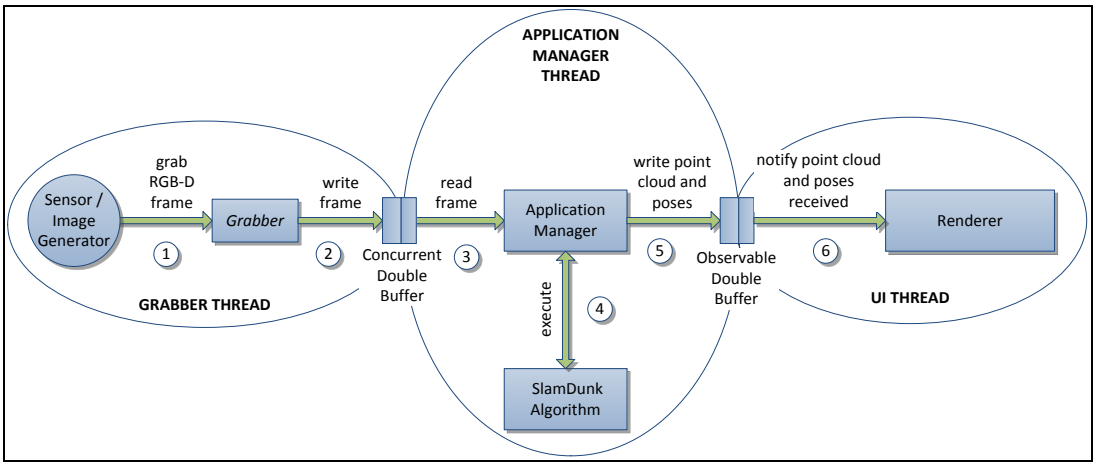

Figura 4.1: Schema generale della struttura di SlamDunk su Android

Questo schema potrebbe anche essere visto come un sequence diagram UML rappresentante il livello di astrazione più alto dell'applicazione, ovvero consistente nelle chiamate a procedura di più alto livello.

I passaggi indicati in figura, indicanti una singola iterazione dell'algoritmo, verranno ora spiegati uno ad uno:

1) Il porting prevede la presenza di una entità *Grabber* di tipo interfaccia (a ciò si deve il nome in corsivo). Questa entità ha il compito di astrarre dal singolo componente che andrà ad effettuare l'operazione di estrazione del frame RGB-D. In tal modo, si potranno avere differenti implementazioni del grabber, selezionabili anche a runtime qualora fosse opportuno. Ciò è stato realizzato anche in previsione di eventuali realizzazioni del grabber su diverse tipologie di sensori, non necessariamente vincolate all'utilizzo di OpenNI e perciò presentanti codice spesso molto differente fra di loro.

   Il grabber così definito si interfaccerà al sensore per ottenere i dati del frame RGB-D correntemente visualizzato dallo stesso. È prevista anche la possibilità di effettuare grabbing a partire da immagini, soprattutto a scopo di test, le quali in questo caso verranno semplicemente lette dalla memoria non volatile del dispositivo mobile in uso;

2) Una volta ottenuto il frame RGB-D, lo stesso verrà scritto all'interno di un Concurrent Double Buffer. Il formato del frame non è importante ma si prediligerà l'utilizzo di array di byte per RGB e float per depth. La tipologia di double buffer utilizzata è in grado di operare fra diversi thread in esecuzione concorrente. Per quanto concerne questa operazione di scrittura, essa non è in alcun modo bloccante ed anzi sblocca gli eventuali thread in attesa di dati sul buffer, qualora esso fosse stato precedentemente vuoto.

   Il funzionamento di un classico double buffer prevede la presenza di due dati simultaneamente, uno (il meno recente) inserito nel front buffer ed un altro (il più recente) inserito nel back buffer. Il prelievo avviene recuperando il dato presente nel front buffer e trasferendo, se presente, all'interno del front buffer il dato precedentemente inserito nel back buffer. L'inserimento invece avviene sul front buffer qualora non vi siano dati in esso, altrimenti avviene sul back buffer. Questa tipologia di buffer è in grado di mitigare le problematiche dovute alla diversa velocità di esecuzione dei due moduli (Grabber ed Application Manager);
3) In questa fase l'Application Manager legge il frame dal Concurrent Double Buffer. L'Application Manager altro non è che un modulo che esegue ciclicamente le stesse operazioni, iniziando appunto da una lettura dal buffer. Per evitare una lettura a polling (ovvero ripetendo il ciclo in caso il dato non sia presente e richiedendo costantemente il nuovo dato, operazione che utilizza inutilmente la CPU), si è deciso di implementare una lettura bloccante per il Concurrent Double Buffer: all'atto di lettura del frame, se esso è presente verrà immediatamente restituito all'Application Manager, in caso contrario il thread su cui è in esecuzione tale modulo verrà sospeso fino all'arrivo dei dati;
4) In seguito alla ricezione del frame RGB-D, l'Application Manager invia lo stesso in input all'algoritmo SlamDunk, il quale adotterà le

operazioni descritte nel Capitolo 2, restituendo in output una matrice della posa ed un enumerativo atto ad indicare il risultato dell'operazione (fallimento, frame tracked o keyframe tracked). Questa fase risulta piuttosto dispendiosa in termini di tempo di esecuzione e potrebbe rallentare il prelievo di un nuovo frame RGB-D, operazione che potrà avvenire soltando in seguito all'esecuzione dell'algoritmo.

Se la velocità del modulo qui considerato, rispetto alla velocità del Grabber, fosse sicuramente inferiore, si potrebbe pensare, per semplicità, di inviare il frame RGB-D direttamente all'Application Manager dal Grabber, senza passare per il double buffer precedentemente descritto;

5) Ottenuto il risultato dall'algoritmo, qualora esso rappresenti l'individuazione di un nuovo keyframe, alcuni dati verranno inseriti in un successivo double buffer. Innanzitutto verrà proiettata l'immagine RGB nello spazio 3D, operazione compiuta dall'Application Manager, ottenendo così una point cloud che verrà inserita nel double buffer, unita alla posa appena ricevuta ma soltanto in seguito ad una sua trasformazione (descritta nella Sezione 4.4 dell'attuale Capitolo). Inoltre, nello stesso buffer si inserirà anche una lista delle pose che sono variate in seguito all'ottimizzazione del grafo, dopo essere state anch'esse trasformate.

   Il modo in cui i dati vengono aggiornati è favorito dalla presenza di una struttura dati all'interno del Renderer, la quale memorizzerà le point cloud ricevute associando ognuna ad un preciso timestamp. La lista delle pose che variano in seguito all'ottimizzazione è perciò composta da una coppia di valori: la matrice di posa da una parte e il timestamp della point cloud associata dall'altra. In questo modo è possibile accedere facilmente ai dati necessari, tramite semplice comparazione dei timestamp;
6) Il double buffer in cui i dati sono stati inseriti non è lo stesso indicato in precedenza, bensì è una struttura dati chiamata Observable Double

Buffer. Questo tipo di buffer notifica i thread Observer associati qualora vi siano state delle modifiche al proprio contenuto. Ciò risulta utile in questo caso, ma anche su altre piattaforme, in quanto si preferisce aggiornare il thread della UI soltanto se vi sono state delle modifiche allo scenario. In caso contrario, il thread eseguirebbe lo stesso codice di visualizzazione svariate volte, occupando CPU inutilmente e rallentando di conseguenza l'esecuzione di altre applicazioni o dell'algoritmo stesso. Inoltre, non sarebbe possibile utilizzare lo stesso buffer precedente, in quanto bloccare il thread della UI bloccherebbe l'applicazione, rendendola inutilizzabile.

In seguito alla notifica, la UI verrà aggiornata inserendo la nuova point cloud nella scena 3D ed aggiornando le pose riferite alle point cloud ottimizzate. Da questa operazione si può notare come, alla fine, il Renderer sia il modulo che andrà ad occupare la maggior parte della memoria volatile, in quanto conterrà le diverse point cloud da visualizzare, strutture dati formate da un numero piuttosto elevato di punti.

Viste le caratteristiche stesse che l'applicazione deve soddisfare in termini di performance, questa tipologia di schema risulta ottimale ad evitare un consumo elevato di risorse e si presenta come una delle soluzioni necessarie a garantire un funzionamento adeguato dell'applicativo.

Alcune cose vanno notate per quanto riguarda lo schema appena proposto.

Innanzitutto, tutti e 3 i thread mostrati eseguono il loro codice in modo ciclico: il Grabber Thread preleva frame RGB-D alla velocità con cui li riceve dal sensore; l'Application Manager Thread preleva tali frame ed esegue su di essi SlamDunk, dipendendo perciò dalla velocità dell'algoritmo; l'UI Thread si occupa di visualizzare le entità che sono presenti sullo schermo e gestisce l'interazione con l'utente.

In secondo luogo, in questo caso Application Manager e Renderer vengono intesi come classi Java, sebbene sia evidente la possibilità di avere diverse tipologie di Renderer e diversi Application Manager con politiche differenti

in base alle esigenze. Data la natura specifica dell'applicazione, per ora tali astrazioni sono state tralasciate. Rimane comunque il fatto che, se in futuro vi fosse la necessità di effettuare tale aggiunta, le modifiche risulterebbero piuttosto limitate.

Nei prossimi sezioni verranno analizzate in maggior dettaglio le implementazioni dei vari moduli, concentrandosi prevalentemente sull'algoritmo SlamDunk, il quale rappresenta sicuramente la parte di complessità più elevata nel sistema.

4.3 Grabber

L'entità Grabber è una interfaccia utile ad astrarre lo specifico metodo con cui viene prelevato il frame RGB-D. Tramite essa sarà possibile cambiare tipologia di Grabber con estrema facilità ed aggiungerne di nuovi quando necessario. La Figura 4.2 in seguito mostra il diagramma UML delle classi con le due entità che in questo caso si è deciso di implementare. Alcuni metodi accessor e costruttori di tali entità sono stati tralasciati.

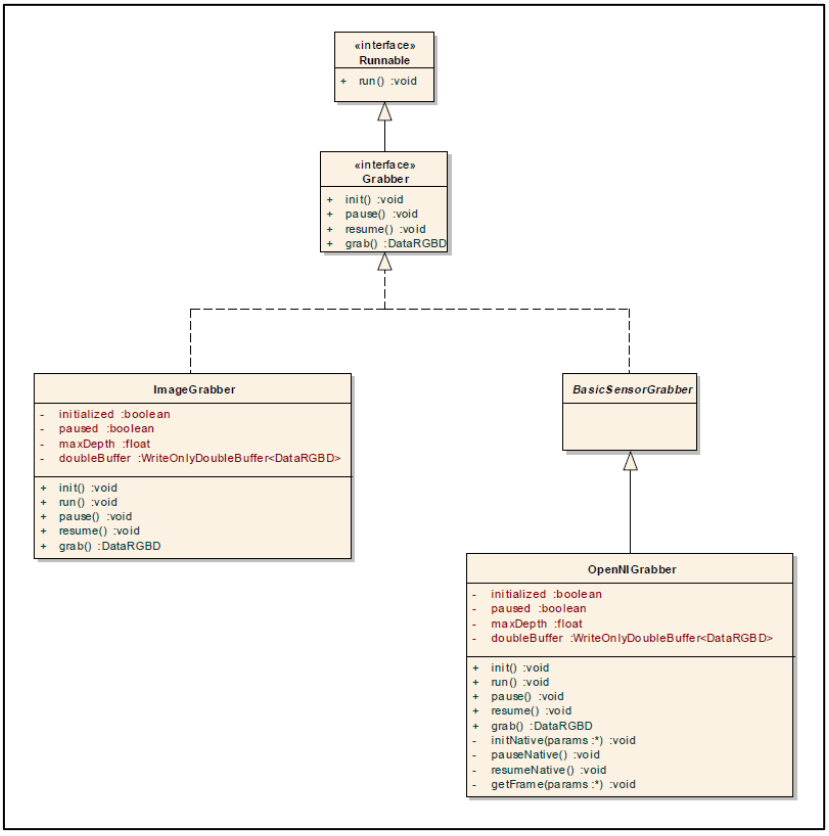

Figura 4.2: Diagramma delle classi Grabber e derivati

L'interfaccia Grabber estende l'interfaccia Runnable presente in Java, tramite cui è possibile definire il Grabber come un thread e metterlo così in esecuzione indipendentemente dal thread principale. Ciò avverrà di solito all'atto di avvio dell'applicazione, comportando l'inizializzazione del Grabber tramite l'invocazione del metodo *init()* e la successiva esecuzione dello stesso come thread, operazione che porterà all'invocazione del metodo *run()* del Grabber. Oltre a ciò sarà possibile sospendere il Grabber durante l'esecuzione e riprenderlo in seguito. Il metodo principale di tale entità è comunque il metodo *grab()*, tramite cui è possibile ottenere i dati dalla sorgente associata, siano essi dati provenienti da immagini su file oppure da sensori 3D reali. Tale metodo verrà continuamente invocato dalla stessa entità qualora si sia posto in esecuzione il thread, sebbene tale implementazione potrebbe variare di caso in caso, venendo implementata soltanto nelle classi sottostanti. È anche possibile non avviare il Grabber come thread, ma inizializzarlo normalmente ed invocare il metodo di grabbing su di esso quando necessario. In tal caso non sarà obbligatorio l'utilizzo del double buffer per il trasferimento dei dati, ad esempio nel caso si prevedesse di eseguire il metodo di grabbing all'interno dell'Application Manager.

Passando alle due classi che implementano l'interfaccia Grabber, va notata la presenza del WriteOnlyDoubleBuffer, il quale non è altro che un'astrazione di un double buffer write-only, in modo tale da permettere l'accesso in sola scrittura al buffer, astraendo anche dai dettagli implementativi della soluzione. Il buffer vero e proprio, come è già stato discusso in precedenza, è un ConcurrentDoubleBuffer date le specifiche esigenze di questa applicazione. Ciò non toglie che in seguito possano essere ideate implementazioni diverse del buffer, se non addirittura un cambio nella logica applicativa, comportando in tal caso modifiche marginali nelle classi che implementano il grabber, essendo esse dipendenti da astrazioni.

Per quanto riguarda l'ImageGrabber vi è poco altro da aggiungere. Il metodo grab() legge le immagini da file e le restituisce sotto forma di oggeto DataRGBD, sulla cui implementazione non è necessario soffermarsi vista la relativa semplicità nella rappresentazione dei dati memorizzati. I restanti metodi seguono le linee guida descritte per l'interfaccia.

Per quanto riguarda l'entità OpenNIGrabber vi sono alcuni elementi da sottolineare. I metodi di inizializzazione, pausa, ripresa e l'operazione di grabbing si presentano differenti, appoggiandosi ad alcuni metodi privati, rispettivamente *initNative()*, *pauseNative()*, *resumeNative()* e *getFrame()*, i quali non sono altro che metodi nativi collegati a codice C/C++ interfacciato al sensore specifico. In tal modo è possibile controllare il sensore tramite l'utilizzo di questa classe. I parametri dei metodi di inizializzazione e grabbing sono stati tralasciati in quanto strettamente dipendenti dall'implementazione. Essi possono essere parametri in ingresso ma anche in uscita e si sottolinea il fatto che l'utilizzo di dati primitivi è consigliato nel caso di invocazione di metodi nativi, in quanto il passaggio di un intero oggetto produce un overhead decisamente maggiore. Il codice C/C++ farà uso delle API OpenNI qualora si utilizzino i sensori Xtion Pro Live o Kinect. Per sensori di altro tipo sarà necessario realizzare codice che si interfacci ai driver del nuovo sensore ed eventualmente realizzare un'ulteriore classe per ogni differente sensore. Spesso i driver saranno realizzati in codice C/C++, per cui risulta quasi ovvia la presenza di metodi nativi in ogni sensore specifico.

L'utilizzo della classe astratta BasicSensorGrabber fornisce maggiore chiarezza nella gerarchia dei grabber, distinguendo così quelli dipendenti da uno specifico sensore rispetto agli altri. Tale classe astratta non risulta comunque strettamente necessaria, non fornendo l'implementazione di alcun metodo così da lasciare maggiore libertà alle realizzazioni sottostanti.

4.4 Application Manager

Il modulo centrare dell'applicazione è dato dall'Application Manager, il quale fa da intermediario fra i dati ricevuti in input dal grabber e l'output risultante di visualizzazione sulla scena 3D. L'Application Manager viene realizzato come thread separato ed utilizza i dati RGB-D inviandoli all'algoritmo SlamDunk per poi manipolare l'output dello stesso.

Le modalità di funzionamento dell'Application Manager sono già state sintetizzate in precedenza. Qui di seguito verrà mostrata la logica di questo componente (Figura 4.3) in modo più dettagliato rispetto allo schema generale, facendo uso di un diagramma UML degli stati.

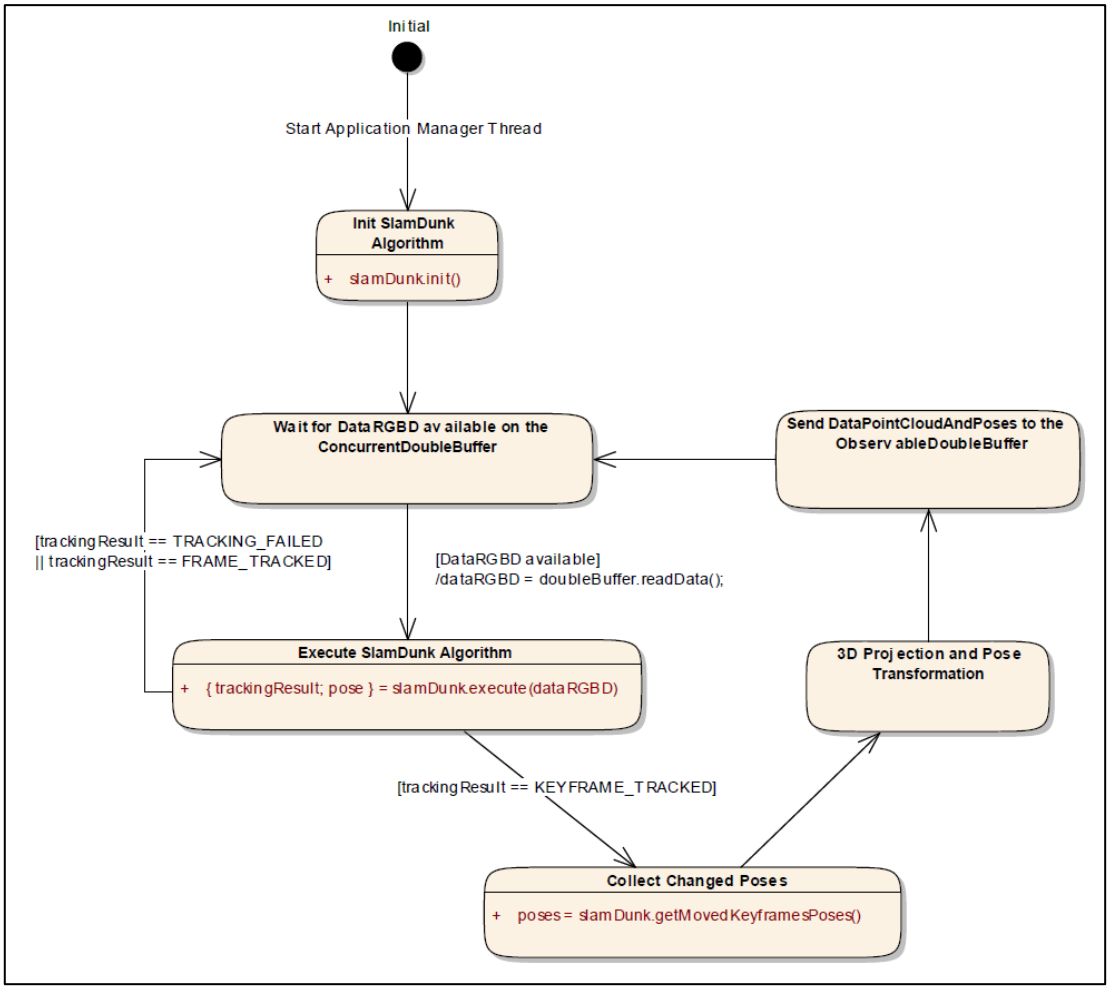

Figura 4.3: Diagramma degli stati dell'Application Manager

Buona parte di questa logica è già stata spiegata in precedenza, compreso l'aggornamento della scena soltando all'atto di ottenimento di un nuovo keyframe.

Nella soluzione attuale si è deciso di inizializzare l'algoritmo SlamDunk all'interno dell'Application Manager, prima di iniziare a ricevere i primi dati. L'utilizzo dei double buffer in ambo le vie aiuta a disaccoppiare il tutto facilitando la fluidità di esecuzione ma soprattutto permettendo di cambiare l'implementazione di uno dei tre moduli senza che ciò vada ad impattare sull'altro. L'unica restrizione riguarda la tipologia di dati scambiata attraverso i buffer. Una ulteriore particolarità riguarda l'impostazione in sola lettura per il double buffer proveniente dal sensore ed in sola scrittura per il double buffer diretto verso il modulo di rendering. In tal modo si evita la possibilità di utilizzare operazioni di fatto incoerenti con la logica a moduli dell'applicazione.

Una cosa che precedentemente non era stata dettagliata riguarda lo stato di proiezione 3D dell'immagine e la trasformazione delle pose. La proiezione 3D avviene in modo simile a quanto illustrato nel capitolo relativo a SlamDunk, con alcuni accorgimenti. Dato un pixel in coordinate *(u, v)*, il punto proiettato di conseguenza viene in questo caso espresso in coordinate omogenee come:

$$p_{so} = \begin{pmatrix} u * D \\ v * D \\ D \\ 1 \end{pmatrix}$$

Anzichè attuare la moltiplicazione con K^{-1}, tale matrice viene trasformata attraverso l'aggiunta di una riga ed una colonna allo scopo di rendere possibile la moltiplicazione con il punto:

$$K_o^{-1} = \begin{pmatrix} K^{-1} & \overline{0} \\ \overline{0} & 1 \end{pmatrix}$$

Ricordando che la posa è rappresentata da *T*, l'ottenimento del punto finale in coordinate omogenee può essere ottenuto tramite l'operazione $p_o = T * K_o^{-1} * p_{so}$. Grazie a questa rappresentazione si ottiene facilmente la trasformazione della posa, data da:

$$T_f = T * K^{-1}$$

I punti delle point cloud risulteranno quindi delle proiezioni dell'immagine seguendo la formula con cui si è ottenuto p_s in coordinate omogenee, mentre le pose verranno trasformate utilizzando la formula precedente con cui si ottiene T_f.

La motivazione di questo accorgimento è meglio comprensibile ragionando sulla fase di rendering dell'applicazione, in cui la point cloud definita dovrà subire in ogni caso una trasformazione che coinvolge i propri punti, in modo tale da venire rappresentata tramite coordinate coerenti con le impostazioni OpenGL scelte. È possibile approfittare di questa trasformazione per accorpare in essa le matrici di posa e della camera presenti nell'applicazione, in modo tale da evitare l'esecuzione di trasformazioni sui punti per due volte di seguito, il che non farebbe altro che diminuire le performance. Inoltre, non è necessario memorizzare i punti delle point cloud in coordinate omogenee in quanto, all'atto della trasformazione, essi verranno automaticamente estesi da OpenGL.

Queste considerazioni, apparentemente al di fuori dei compiti concessi all'Application Manager, evitano di fatto una moltiplicazione dei punti della point cloud. La matrice T_f così generata rappresenta la model matrix della point cloud calcolata.

4.5 Renderer

Per la visualizzazione di scene 3D Android si basa sull'utilizzo di OpenGL ES [K10], ovvero un sottoinsieme delle librerie OpenGL adattato per i dispositivi integrati. Nell'implementazione del Renderer è stata utilizzata la versione 2 di OpenGL ES, la quale presentava diverse novità interessanti in termini di personalizzazione. Da questa versione è infatti possibile un'efficace personalizzazione degli shader, in modo tale da sviluppare concretamente i diversi passaggi della pipeline grafica, ottenendo un maggiore controllo su quanto viene effettuato e sulle performance raggiunte. Dato che la gestione della interfaccia grafica non rappresenta lo

scopo principale di questa tesi, essa verrà descritta brevemente per poi passare al porting vero e proprio dell'algoritmo SlamDunk. In tal modo si è anche in grado di completare la trattazione dei moduli puramente Android di questa tesi, di supporto all'algoritmo vero e proprio nonchè necessari per ottenere un'applicazione adatta e performante.

La User Interface realizzata permette di visualizzare le point cloud su schermo, nonchè di navigare nello scenario 3D attraverso il touch screen del dispositivo. Sono disponibili le funzionalità di zoom, traslazione e rotazione all'interno della scena, realizzate modellando un oggetto Camera che fornisca il punto di vista attuale all'interno della scena, dato dai vettori lookAt ed up. Va notato come tali funzionalità richiedano operazioni su vettori o matrici, le quali sono state realizzate in Java all'interno dell'applicazione, senza dipendere ulteriormente da alcuna libreria esterna. Le point cloud sono state realizzate specificando la posizione dei punti ed il loro colore, facendo uso dei Vertex Buffer Object (VBO) per un rendering che possa essere il più efficiente possibile. Come già precedentemente indicato, inoltre, la scena 3D verrà ridisegnata solamente quando necessario, ovvero durante il movimento della Camera e durante l'aggiunta di una nuova point cloud. È presente la possibilità di mettere in pausa e riprendere il capturing (da un menu), la quale si riflette nelle operazioni del Grabber specifico in utilizzo.

Entrando maggiormente nello specifico va ricordato che il Renderer si interfaccia ad un double buffer dopo essersi aggiunto come suo Observer, seguendo l'omonimo design pattern. In tal caso, in seguito alla scrittura da parte dell'Application Manager, avviene una segnalazione che indica la necessità di ridisegnare la scena 3D. Ricevuta tale segnalazione il thread della UI procederà il prima possibile ad effettuare l'operazione. Questa impostazione, che consente l'aggiornamento della UI soltanto in seguito alle modifiche, può essere attivata direttamente in Android senza comportare la necessità di una sua implementazione diretta.

In ambiente Android il Renderer è inserito all'interno di una entità chiamata GLSurfaceView, la quale rappresenta una particolare area sullo schermo del

dispositivo. Tale entità si interfaccia al Renderer per eseguire le diverse operazioni di visualizzazione del scena. Essa ha inoltre il compito principale di gestione dell'input, il quale si rifletterà su uno specifico movimento della Camera.

Una aggiunta opzionale nel Renderer è data dalla presenza, all'interno dell'interfaccia, dell'immagine attualmente trasmessa dal sensore e manipolata dall'algoritmo. Questa aggiunta si presenta utile per comprendere appieno le immagini effettivamente analizzate dall'applicazione. Per stimare la velocità dell'applicazione è stato anche aggiunto un misuratore degli FPS.

La Figura 4.4 mostra uno screenshot dell'applicazione, per meglio comprendere i dettagli dell'interfaccia grafica realizzata. Si può notare la presenza dell'immagine attualmente campionata in basso a sinistra, le point cloud analizzate al centro ed il numero di FPS in alto a destra. Ulteriori modifiche sono possibili per ottenere miglioramenti visivi, sebbene non facciano parte dello scopo di questo lavoro di tesi.

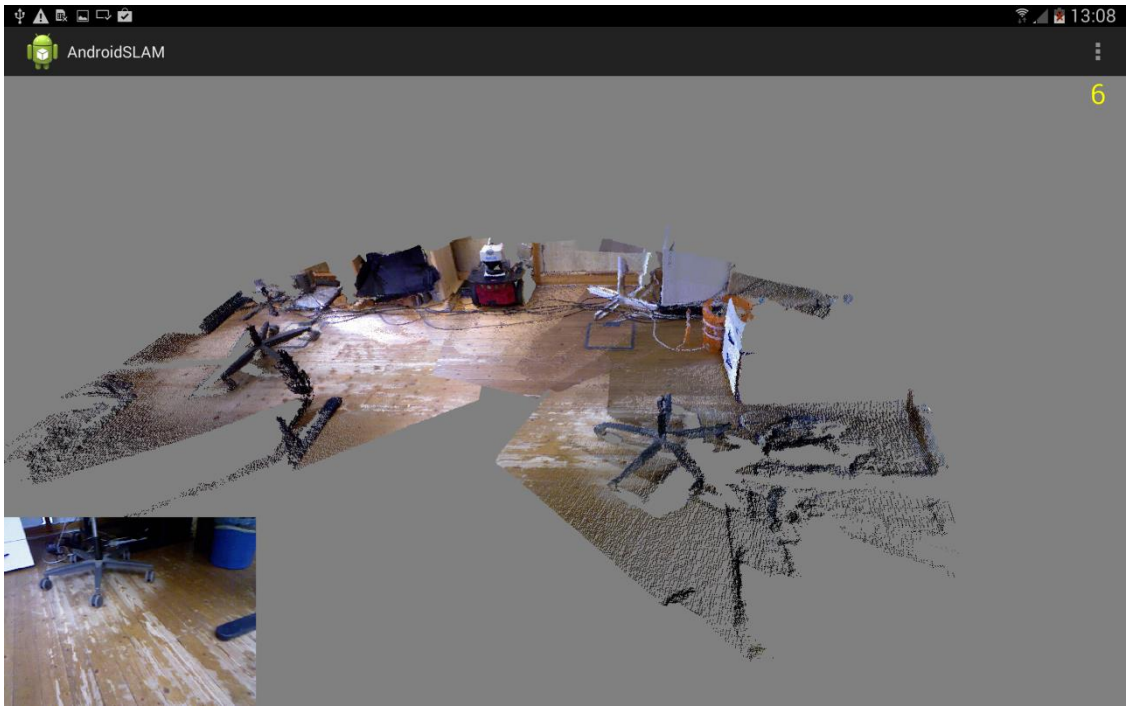

Figura 4.4: User Interface di SlamDunk su Android

4.6 Algoritmo SlamDunk su piattaforma Android

Avendo analizzato i moduli di appoggio allo sviluppo dell'algoritmo su piattaforma Android, si può ora passare all'implementazione vera e propria, la quale ha seguito due strade principali diverse.

Una strada prevede il porting quasi totale del codice C++ in codice Java, fatta eccezione per il codice dipendente da librerie esterne non sostituibili da codice Java, quali ad esempio OpenCV (keypoint detection, feature extraction e matching) e G2O (local optimization). Tale procedura è sicuramente più lenta da sviluppare e potrebbe causare un certo calo di performance, ma rende il progetto di più facile estensione e modificabilità in un futuro, permettendo anche un debugging più semplice.

La seconda strada prevede invece il mantenimento della maggior parte del codice in C++ adattandolo al dispositivo mobile e tentando perciò di sfruttare la possibilità di compilazione di buona parte delle librerie esterne. Questo approccio facilita sicuramente lo sviluppo e migliora le performance, ma comporta una maggiore difficoltà nella modifica del codice e nel suo debugging.

Scegliendo di procedere con ambo gli approcci si tenta di mitigare gli svantaggi di entrambi, rimandando alla versione Java nel caso di modifiche future e mantenendo la versione C++ nei casi che necessitano maggiore velocità di esecuzione.

Si partirà ora dall'analisi della versione C++ di SlamDunk, indicando innanzitutto lo schema software della stessa, maggiormente dettagliato rispetto allo schema logico dell'algoritmo indicato nel Capitolo 2. Dopodichè si passerà alla versione Java dell'applicativo, sottolineandone le differenze. Infine, si indicheranno le modifiche apportate indipendentemente dalla versione dell'applicazione.

4.6.1 Versione C++

La versione C++ dell'algoritmo si presenta molto simile a quella desktop, pur introducendo alcune modifiche, soprattutto per quanto concerne il

metodo di matching utilizzato. Rimangono perciò valide tutte le dipendenze precedentemente individuate, utilizzabili su Android attraverso l'NDK senza problemi.

Nella sezione attuale verrà analizzata la struttura finale dell'algoritmo ed il modo in cui interagiscono i diversi componenti, mentre le considerazioni in merito alle performance ed ai diversi algoritmi utilizzabili verranno date in seguito.

La Figura 4.5 mostra il diagramma UML delle classi riferito prevalentemente al tracking della camera, pur contenendo una parte del modulo di mapping dato dalla presenza del quadtree e da un suo utilizzo nel diagramma.

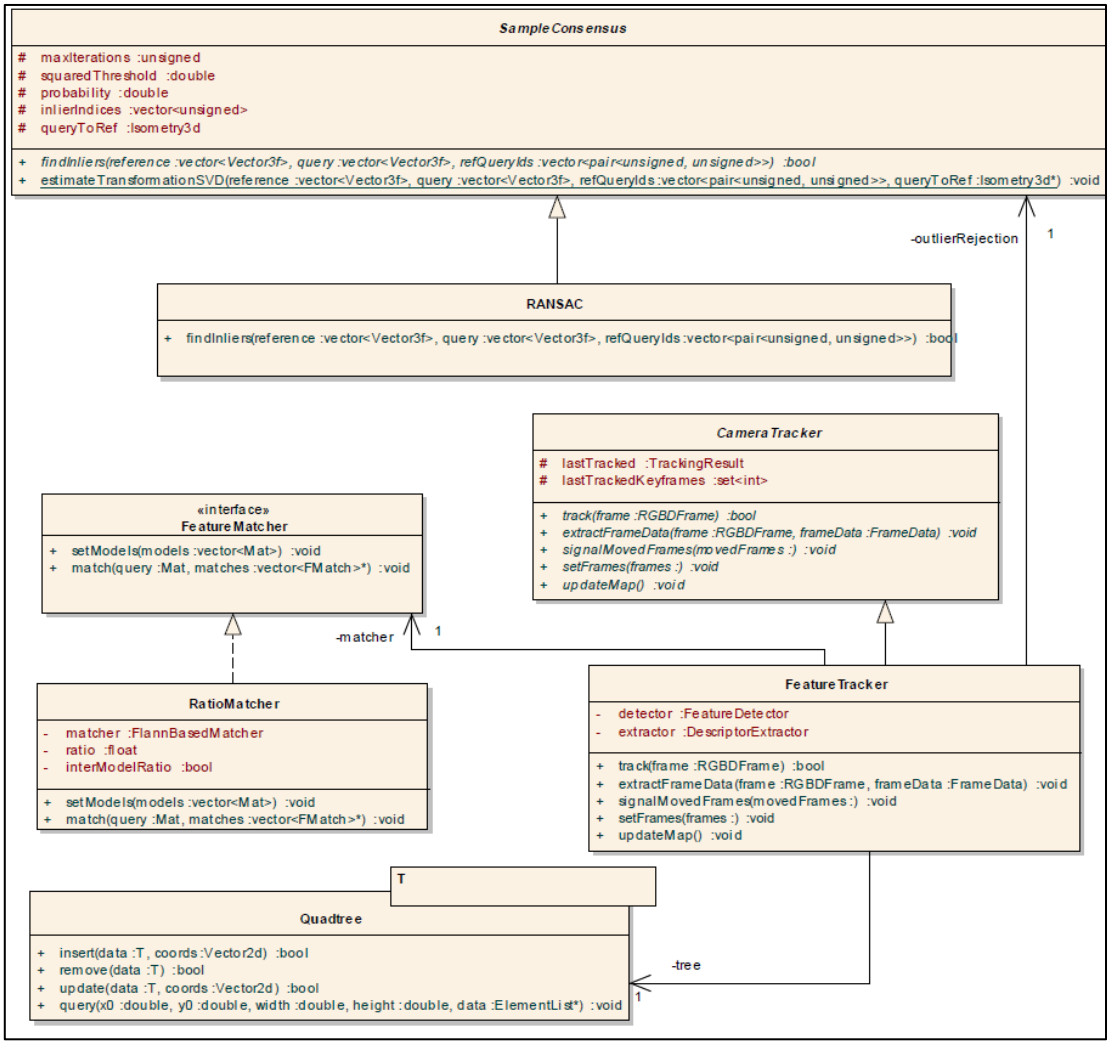

Figura 4.5: Diagramma delle classi del modulo di tracking

La classe principale, la quale si appoggia alle restanti classi del diagramma, è il *FeatureTracker*, estensione della classe astratta *CameraTracker* di cui ne implementa i metodi. Questa classe ha il compito principale di effettuare il tracking (metodo *track()*), costituito dai seguenti passaggi:

1) A partire da un frame RGB-D vengono estratte le features, utilizzando le classi *FeatureDetector* e *DescriptorExtractor* di OpenCV. Tali features sono quindi inserite in una struttura dati chiamata *FrameData*, assieme ai keypoints già proiettati in 3D, da utilizzare per le future operazioni di rimozione degli outlier. Il tipo di algoritmo utilizzato per l'estrazione delle features può essere modificato a piacimento sia per quanto riguarda la keypoint detection che per la feature extraction, essendo le classi *FeatureDetector* e *DescriptorExtractor* soltanto delle generalizzazioni dello specifico algoritmo. L'operazione di estrazione viene attuata tramite il metodo *extractFrameData()*;

2) In seguito all'estrazione si effettua il matching fra le features appena estratte e quelle del pool. Ciò avviene appoggiandosi all'interfaccia *FeatureMatcher*, la cui attuale implementazione è realizzata dalla classe *RatioMatcher*. Il matching è effettuato dall'operazione *match()* ed effettua anche il filtering tramite ratio test. Anche in questo caso, l'unico vincolo riguarda l'utilizzo della classe di OpenCV *FlannBasedMatcher*, che essendo generica lascia la possibilità di utilizzare differenti indici implementati in FLANN. Il metodo *setModels()* è utile ad impostare il pool di features (le features di ogni immagine sono memorizzate all'interno di una struttura dati Mat di OpenCV), essendo lo stesso variabile in base alla posizione del dispositivo mobile. Tale impostazione avviene in seguito all'operazione di query sul quadtree, in grado di aggiornare i dati attualmente in uso. L'aggiornamento non viene effettuato dal *FeatureTracker* in modo autonomo, bensì sotto richiesta in seguito all'invocazione dei metodi *updateMap()*, *signalMovedFrames()* o *setFrames()*;

3) Dopo il matching, l'operazione di filtering tramite RANSAC si appoggia alla classe astratta *SampleConsensus*, a sua volta estesa dalla classe concreta *RANSAC*. Il metodo *findInliers* accetta come parametri due array contenenti i dati proiettati per quanto concerne i punti dell'immagine corrente e quelli delle immagini con cui è avvenuto il match. I tipi di dato con cui si rappresentano i punti derivano dalla libreria Eigen. Oltre a questi due parametri è richiesto un array, chiamato in questo caso *refQueryIds*, contenente le coppie di indici reference/query, essendo le stesse probabilmente memorizzate nei due array in indici diversi. L'algoritmo di RANSAC procederà nella scelta di 3 numeri random differenti fra di loro, usando gli stessi come indici nell'array reference/query e raccogliendo così i 2 indici che facciano riferimento agli altri 2 array (tutto ciò per tutti e 3 i numeri random). Dopo l'esecuzione del metodo *findInliers*, si procederà ad ottenere gli indici dei punti sopravvissuti al filtering (variabile *inlierIndices* di *SampleConsensus*) e ad invocare il metodo statico *estimateTransformationSVD* per il calcolo della posa finale, tenendo in considerazione soltanto gli inliers selezionati. Il risultato è inserito nel parametro *queryToRef*, il quale è rappresentato da una Isometry3d, una struttura dati di Eigen che indica una matrice di posa, con valori di rotazione e traslazione;

4) Ottenuta la posa, la stessa verrà inserita nella struttura dati di tipo *TrackingResult*, contenente diversi dati utili all'applicazione: il FrameData, i matches effettuati e l'overlapping score ottenuto, utile per la selezione del keyframe.

Verrà ora analizzata la seconda parte del diagramma delle classi, mostrato in Figura 4.6, nel quale si può notare l'interfacciamento che si ha con il tracker e la gestione delle operazioni di ottimizzazione e mapping.

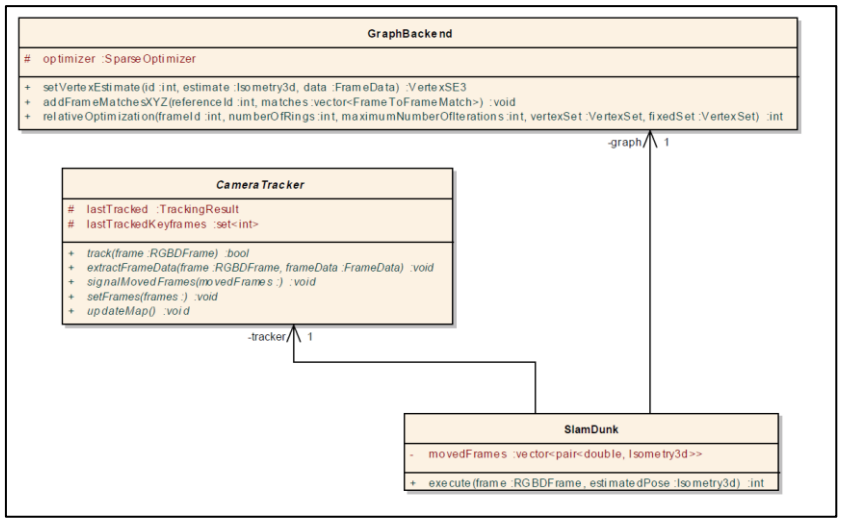

Figura 4.6: Diagramma delle classi sul resto del sistema

Il questa parte del diagramma delle classi è possibile notare la presenza della classe principale dell'intero algoritmo, chiamata *SlamDunk*, la quale viene invocata specificando il frame RGB-D e la posa parametro di tipo out. Tale metodo inoltre restituisce un intero che indica il tipo di risultato (tracking fallito, frame tracked, keyframe tracked). Essendo tale classe realizzata in C++, essa verrà inizializzata e successivamente invocata tramite JNI, con opportuna trasformazione dei dati in ingresso (frame RGB-D) ed in uscita (intero del risultato, posa e pose ottimizzate in caso di individuazione di un nuovo keyframe). Effettuando tutto il resto del codice in C++ e richiedendo perciò una singola trasformazione dei dati, è chiaro come tale soluzione sia maggiormente performante, causando un overhead limitato.

In seguito all'invocazione del metodo di esecuzione di *SlamDunk*, lo stesso verificherà innanzitutto se l'algoritmo è alla prima iterazione o ad una delle successive. Nel caso fosse alla prima, non si fa altro che estrarre il FrameData (features e keypoints 3D attraverso il metodo *extractFrameData()*) e impostare il frame corrente come principale attraverso il metodo *setFrames()*, il quale di conseguenza aggiorna il quadtree presente nel tracker. Nel caso di una qualsiasi iterazione successiva, si procede invocando il metodo principale del *CameraTracker*, chiamato appunto *track()* e precedentemente descritto in tutte le sue

operazioni. In seguito all'invocazione di tale metodo si hanno a disposizione informazioni sulla posa ottenuta e sul valore di overlapping, il quale verrà confrontato con il valore soglia dell'applicazione. Se l'overlapping è inferiore alla soglia, si procederà con l'ottimizzazione, attuata dalla classe *GraphBackend* (metodo *relativeOptimization()*), non prima di aver aggiunto il frame al grafo attraverso il metodo *addFrameMatchesXYZ()*. L'ottimizzazione prevede di specificare il numero massimo di livelli di ricerca nel grafo, tramite cui l'algoritmo calcolerà il *vertexSet* (nodi del grafo da ottimizzare) ed il *fixedSet* (nodi del grafo al confine e quindi mantenuti fissi), parametri in uscita al metodo *relativeOptimization()*, utili per aggiornare le pose che sono state modificate. Le ripercussioni dell'ottimizzazione si rifanno alla variabile *movedFrames* della classe *SlamDunk* la quale tiene traccia, ad ogni iterazione, dei diversi keyframes che si sono mossi rispetto all'iterazione precedente. Tali keyframes verranno prelevati e permetteranno un aggiornamento di diverse pose all'interno dell'interfaccia grafica dell'applicazione. Come ultima cosa, nel caso in cui non sia stato trovato un keyframe, verrà richiesto l'aggiornamento della posizione del quadtree (metodo *updateMap()*) qualora il valore di shift del frame corrente sia superiore ad una certa soglia (come già spiegato nel Capitolo 2).

Il diagramma di sequenza successivo (Figura 4.7) elenca meglio le diverse invocazioni effettuate dall'algoritmo, partendo dalla classe di più alto livello, ovvero SlamDunk. In questo caso la variabile *result* è un boolean atto ad indicare il risultato del tracking, in quanto l'operazione può fallire per diversi motivi, ad esempio un numero di features estratte troppo basso. La variabile restituita al termine della sequenza, chiamata *res*, rappresenta invece l'intero che darà indicazioni sull'esito dell'intera invocazione dell'algoritmo.

Va notato come il diagramma faccia riferimento ad un sistema a regime, mentre la sequenza relativa alla prima iterazione è stata tralasciata poichè non risulta particolarmente significativa.

Infine, si può notare come diversi metodi contengano parametri passati come riferimento, in modo tale che negli stessi possa essere contenuto il risultato. Basti vedere il primo metodo richiamato, contenente il parametro *estimatedPose* che cosituirà l'output, oppure nel metodo statico *estimateTransformationSVD()* il parametro *queryToRef*. Nella versione Java dell'applicativo bisognerà tenere in considerazione questa particolarità, allo scopo di evitare confusione nell'utilizzo dei riferimenti.

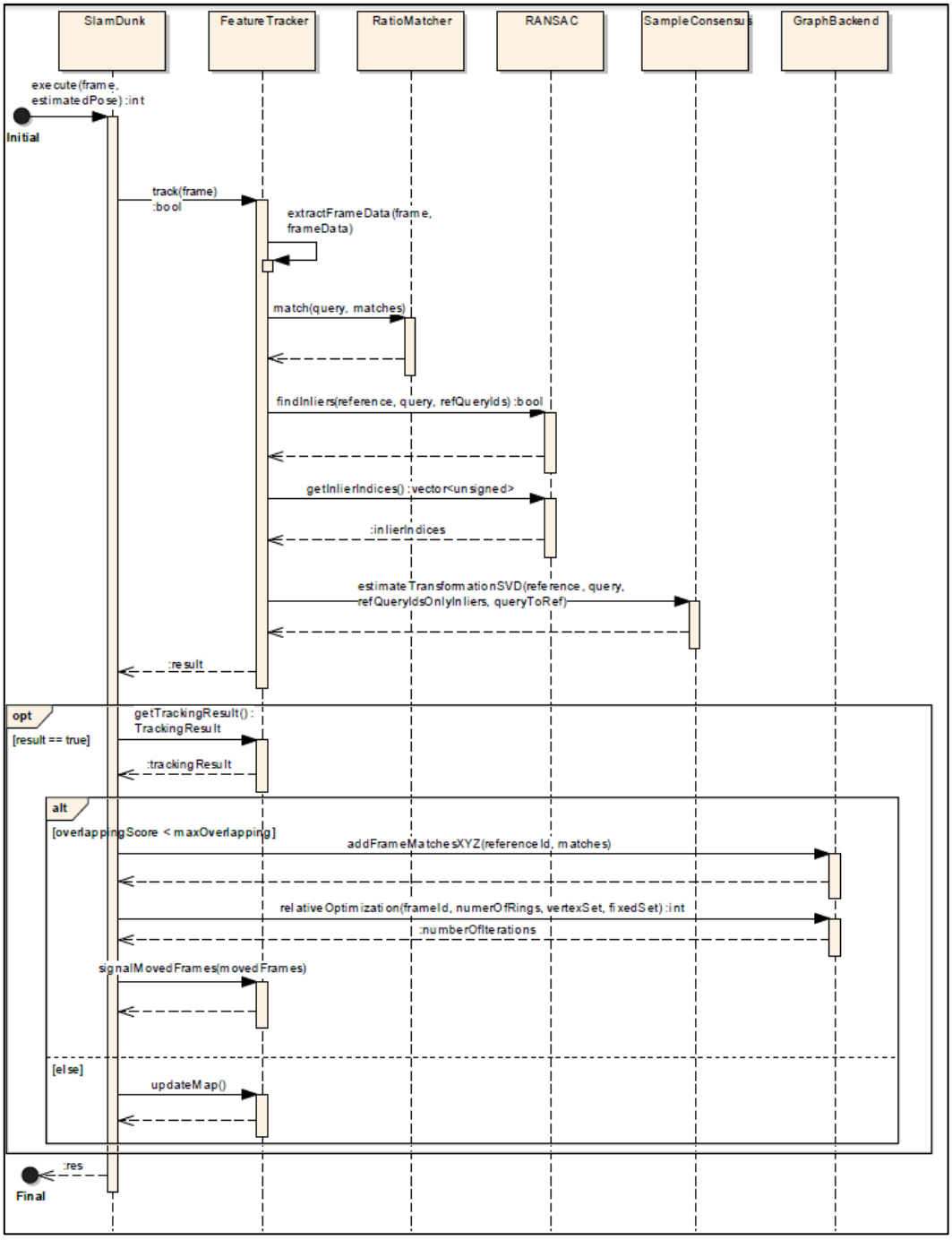

Figura 4.7: Diagramma di sequenza del sistema

4.6.2 Versione Java

Date alcune visibili differenze fra il linguaggio C++ e Java, le due versioni dell'applicativo non potranno essere completamente simili, sebbene presentino notevoli somiglianze.

Qui di seguito si descriveranno le maggiori differenze fra le due versioni, che per il resto presentano un diagramma delle classi molto simile fra di loro.

Come prima considerazione, l'utilizzo della libreria Eigen, così come di tutte le altre librerie C++, non può più avvenire direttamente e deve perciò essere realizzato mediante invocazione di metodi nativi o sostituendo gli opportuni moduli con del codice Java analogo. Si è perciò deciso di sostituire parzialmente Eigen con una alternativa da utilizzare per le operazioni più semplici, rappresentando vettori e matrici e supportando le principali operazioni matriciali, fra cui somma e moltiplicazione. Si è deciso di adottare una soluzione nuova invece di appoggiarsi a librerie Java con l'obiettivo di ottenere un maggiore controllo sul codice adoperato e facilitare il passaggio dagli oggetti Java a C++ quando è necessario, adattando il formato di invio in modo tale che sia il più efficiente possibile. Avendo costruito questa alternativa, sarà possibile trasformare la maggior parte del codice C++ in codice Java evitando l'invocazione di metodi nativi per semplici operazioni matriciali.

Nello specifico, la soluzione prevede il mantenimento dei dati per vettori e matrici all'interno di semplici array Java, facilmente utilizzabili come parametro nei metodi JNI che richiedono l'utilizzo di tali strutture dati. Infatti, alcune parti di codice verranno mantenute in C++, per motivi di comodità ed efficienza, come verrà in seguito indicato. Si potrà perciò passare dagli array Java a strutture dati Eigen, essendo che tale libreria supporta questo tipo di mapping in modo efficiente.

Data la modifica appena descritta, sarà possibile trasformare le classi C++ dei diagrammi in classi Java, mantenendo per la maggior parte dei casi le stesse caratteristiche. Ciò è favorito dal fatto che OpenCV fornisce un

wrapper Java delle principali classi, potendo così mantenere gli stessi nomi e le stesse funzionalità.

Una classe che è stata rivisitata rispetto alla versione C++ è il *Quadtree*, che in tale versione veniva realizzato mediante un uso elevato di tipi generici, utilizzando anche operazioni non facilmente trasformabili in codice Java. Per ovviare a ciò si è deciso di ristrutturare l'albero tramite interfaccia e relativa implementazione, delegando ad una Factory il compito di istanziare l'implementazione desiderata, ricorrendo ai tipi di dato generici soltanto per quanto concerne il contenuto stesso del quadtree.

Invece, le classi che sono state facilmente portate sono: RatioMatcher, FeatureTracker e SlamDunk (con le relative interfacce). Dalle classi elencate sfuggono RANSAC ed il GraphBackend, le quali presentano sì un interfacciamento realizzato in Java, ma il codice dell'implementazione è stato parzialmente mantenuto in C++.

In RANSAC ciò è dovuto alle operazioni effettuate all'interno del ciclo di iterazioni, nello specifico ad alcune dispendiose operazioni di calcolo matriciale come la SVD (Single Value Decomposition) utile al calcolo della trasformazione fra le coppie di punti 3D. Tali operazioni sono state mantenute in codice C++ e richiamate attraverso l'uso di JNI. L'operazione specifica di SVD viene realizzata tramite l'utilizzo di Eigen, dopo aver copiato i dati dalla struttura dati matriciale Java a quella C++ di Eigen. Questo passaggio da codice C++ consente anche di ottenere un vantaggio in termini di tempo di esecuzione, grazie all'appoggio della libreria Eigen.

Per quanto concerne invece il GraphBackend, esso non è nient'altro che un wrapper a quello che si potrebbe considerare come il GraphBackend C++. Tutti i metodi invocati per la modifica del grafo e la sua ottimizzazione si riferiscono a codice C++, il quale utilizza la libreria g2o, compilata su Android, per manipolare il grafo, anch'esso memorizzato come variabile globale nel codice nativo. Per motivi di efficienza, i nodi vengono memorizzati anche nel codice Java, mantenendo sia la posa sia i dati relativi al keyframe (le features estratte e la proiezione in 3D dei keypoints). Questi ultimi dati non sono trasferiti nel grafo C++ per evitare di appesantire il

tutto, soprattutto in quanto non utili per l'operazione di ottimizzazione, essendo le informazioni già memorizzate negli archi del grafo. La lettura delle pose dato l'id del nodo può avvenire direttamente da codice Java senza trasferimento di dati verso codice nativo. Va inoltre notato che in seguito all'ottimizzazione, eseguita nel codice C++, le pose memorizzate in Java verranno aggiornate con i nuovi valori assunti, evitando così problemi di incoerenza fra le due strutture dati. Il livello di ridondanza introdotto è molto utile ad aumentare le performance, aumentando in maniera marginale l'utilizzo della memoria.

Le modifiche apportate non cambiano in alcun modo la qualità dell'algoritmo e sono state verificate per evitare la presenza di errori in fase di programmazione.

La principale differenza fra le due versioni risulta comunque il tempo di esecuzione, il quale è maggiore nella versione Java.

4.6.3 Modifiche indipendenti dalla versione

In questa sessione verranno analizzate diverse modifiche che non dipendono dal linguaggio di programmazione scelto, ma sono conseguenze delle limitazioni date dal contesto di lavoro. La principale tematica qui analizzata riguarda i possibili algoritmi da utilizzare in ambiente mobile, per quanto concerne keypoint detection, feature extraction e matching. Questa scelta influenzerà in modo visibile l'andamento dell'algoritmo, sia in termini di qualità che in termini di tempo di esecuzione. Si cerca perciò il giusto compromesso per raggiungere il migliore rapporto fra questi due importanti parametri.

Tutte le metodologie tenute in considerazione verranno successivamente testate per meglio individuare quelle più adatte.

Scelta del feature detector ed extractor:

La detection dei keypoints e l'estrazione dei descrittori dagli stessi sono due operazioni fra di loro collegate. Tuttavia, ciò non evita l'implementazione e l'utilizzo di algoritmi differenti per queste due fasi. Allo stato attuale infatti esistono algoritmi complessivi, come ad esempio SURF o ORB, in grado di prevedere una fase di determinazione dei punti chiave ed una fase di estrazione dei descrittori. Esistono tuttavia altri algoritmi in cui la fase di detection o estrazione è tralasciata. Nulla vieta, ad esempio, l'utilizzo del detector di ORB unito all'extractor di SURF o viceversa, essendo i due moduli fra di loro separati in modo netto.

Nella scelta del detector e dell'extractor si è tenuto in considerazione prevalentemente il tempo di esecuzione, che deve essere il più ridotto possibile per garantire velocità accettabili su un dispositivo mobile. In secondo luogo va considerata la qualità risultante.

Qui di seguito vengono indicate le configurazioni utilizzate:

- Una prima configurazione prevede l'utilizzo di ORB come primo detector ed extractor, presentando un rapporto discreto fra la velocità di esecuzione e la qualità dell'output dell'algoritmo;
- La seconda configurazione prevede l'utilizzo di ORB nella detection dei keypoints e di BRISK [LSetal11] nell'estrazione delle features, anch'essa in grado di produrre buoni risultati. Non si è in grado di utilizzare il detector di BRISK allo stato attuale, in quanto presenta ancora alcune problematiche nella sua implementazione OpenCV ed i risultati non si sono presentati attendibili;
- L'ultima configurazione analizzata prevede una soluzione personalizzata che fa uso di Upright SURF, il detector di SURF senza il calcolo dell'orientamento dei punti chiave, unito all'estrazione tramite BRISK, con annesso calcolo dell'orientamento. Questa ultima configurazione merita qualche ulteriore dettaglio, siccome il codice di SURF è stato ottimizzato

specificatamente per Android, sfruttando l'utilizzo delle operazioni assembler di ARM NEON.

Per rendere maggiormente comprensibile tale ottimizzazione è utile descrivere brevemente l'operazione di detection effettuata da SURF. L'algoritmo originale inizia ad eseguire ricevendo in input una immagine in formato grayscale, la quale viene trasformata in immagine integrale (anche chiamata *summed area table*). Questo tipo di immagine è formata in questo modo: il valore in posizione *(x, y)* è dato dalla somma dei valori in tutte le posizioni *(i, j)* con $i < x$ e $j < y$. L'equazione che rappresenta il calcolo è la seguente:

$$PI_{xy} = \sum_{i=1}^{x} \sum_{j=1}^{y} PO_{ij}$$

dove PI è il pixel nell'immagine integrale, mentre PO rappresenta il pixel nell'immagine originale.

La trasformazione dell'immagine permette di effettuare efficientemente alcune operazioni di convoluzione sulla stessa. In questo caso il vantaggio più netto si rileva nell'operazione di box filtering. Tale operazione applica un filtro all'immagine con lo scopo di trasformarla e rendere in seguito possibile la rilevazione dei keypoints più significativi. Ciò si attua considerando alcuni pixel all'interno dell'immagine e calcolando l'area fra di essi, in modo facilitato grazie alla presenza dell'immagine integrale. La Figura 4.8 mostra uno dei box filter utilizzati nel caso dell'algoritmo SURF, che ne utilizza un totale di 3, effettuando poi operazioni aggiuntive fra i risultati degli stessi. Dall'immagine si possono notare le aree in bianco e nero, di cui viene calcolata l'area e moltiplicata per il valore presente all'interno (+/- 1), mentre i valori A1, B1 e successivi indicano i pixel dell'immagine utilizzati. A titolo di esempio, il calcolo dell'area in alto a sinistra è così costituito:

$$+1 * (A1 - B1 - D1 + E1)$$

Le frecce nella immagine indicano la direzione in cui il box filter si muove; una prima operazione coinvolgerà i pixel nella posizione 1, l'operazione successiva coinvolgerà quelli alla loro destra in posizione 2 e così via.

L'ottimizzazione effettuata in assembler prevede, tramite l'utilizzo di codice di basso livello, la possibilità effettuare operazioni in contemporanea su più elementi dell'array costituente l'immagine da analizzare. Nello specifico si possono manipolare 4 elementi (4 pixel) alla volta. Ciò permette di ridurre di 4 volte il numero di operazioni effettuate durante il box filtering. Inoltre, questo vale anche nel caso di scaling dell'immagine di 2 o 4 volte (operazione prevista dall'algoritmo SURF), grazie alla possibilità di lettura in interleaving dei dati dell'array.

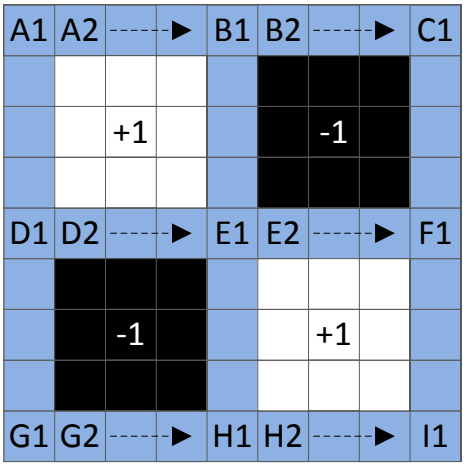

Figura 4.8: Box Filter SURF

Tutte e tre le soluzioni discusse verranno testate nel capitolo successivo, sottolineando pregi e difetti di ognuna di esse, nel tentativo di individuare la soluzione più appropriata.

Scelta del feature matcher:

La scelta dell'algoritmo di matching si presenta molto più semplice, data la presenza di un numero inferiore di alternative e l'impossibilità di utilizzarle tutte con le features scelte. Infatti, gli algoritmi precedentemente scelti per l'estrazione delle features, con conseguente creazione dei descrittori, sono in grado di generare descrittori esclusivamente di tipo binario. Per tali descrittori è conveniente utilizzare soltanto un sottoinsieme degli indici FLANN esistenti, a differenza di quello che era possibile utilizzare per la versione desktop di SlamDunk, appoggiata prevalentemente sull'extractor di SURF.

Nello specifico, si sono testati gli indici LSH (Locality-Sensitive Hashing [LJ07]) e Hierarchical Clustering [ML12]. Le soluzioni sono state testate con i parametri di default, siccome i risultati così ottenuti sono ottimali in svariate situazioni applicative. La soluzione che si è presentata più consona, sebbene non con evidente vantaggio, è quella del Hierarchical Clustering. Ulteriori informazioni sui test realizzati verrano date nel capitolo successivo.

Capitolo 5
Testing dell'applicazione

Dopo aver descritto le parti principali del porting Android dell'applicazione, nonchè definito gli algoritmi più promettenti da poter utilizzare, si passa ora all'attuazione di test che possano mettere in luce la qualità delle strategie utilizzate, sia in termini di performance che in termini di precisione nella ricostruzione 3D.

Invece di concentrarsi sulla differenza fra la versione Java e la versione C++, che in termini di risultati sono analoghe e mostano perciò soltanto una differenza in performance a vantaggio di quest'ultima, i test si sono maggiormente interessati ad individuare gli algoritmi ottimali per l'estrazione dei landmarks ed il loro utilizzo. Alcuni algoritmi sono stati scartati a priori, spesso a causa delle ingenti risorse computazionali richieste (ad esempio l'uso dell'extractor di SURF o SIFT) oppure degli scarsi risultati ottenuti nelle prove effettuate.

La parte iniziale di questo capitolo illustrerà la metodologia utilizzata per effettuare i test e misurarne i risultati. In seguito verranno confrontate le 3 configurazioni indicate nel capitolo precedente, commendando i risultati ottenuti. Successivamente, si effettueranno test più approfonditi sulla configurazione che si è ritenuta più promettente, mostrando inoltre alcuni output dell'algoritmo per quanto concerne la ricostruzione 3D. Infine, il capitolo si concluderà con una panoramica dei sensori che sono attualmente utilizzabili dalla versione Android di SlamDunk, con l'interesse ad ampliare questo insieme.

5.1 Configurazione dei test

I test sono stati effettuati tramite un tablet Samsung Galaxy Tab Pro 10.1 di nuova generazione, in modo tale da permettere un elevato livello di ottimizzazione in un'applicazione che sicuramente richiede un grande numero di risorse computazionali. La tipologia di tablet scelta permette di eseguire le operazioni di ottimizzazione precedentemente implementate, relative all'utilizzo di istruzioni ARM NEON.

La maggiore importanza in questa fase del lavoro di tesi è data a test di natura scientifica, che misurino in modo preciso il margine di errore dell'algoritmo in base a diversi ambienti di esecuzione. Ciò viene svolto per avere un'idea quanto più ampia possibile sui casi in cui l'algoritmo si comporta meglio e quelli in cui ha maggiori difficoltà, senza dare per scontato alcun tipo di risultato. Il lavoro si è perciò basato su alcuni dataset di immagini per effettuare i test [SEetal12], pubblicamente disponibili presso il sito web del dipartimento di Computer Vision dell'Università TUM. Tramite l'utilizzo dell'Image Grabber, vengono lette le immagini RGB e depth e trasferite all'Application Manager per l'esecuzione stessa dell'algoritmo. I risultati in termini di posa vengono salvati sia per quanto riguarda tutti i frame in input all'algoritmo, sia per quanto riguarda i soli keyframes individuati. Al termine dell'esecuzione di un dataset, i dati delle pose così ottenuti vengono confrontati con la traiettoria reale compiuta dal sensore (*groundtruth trajectory*), presente nello stesso dataset in analisi. Il confronto così effettuato permette di ottenere alcuni valori che permettono di comprendere il grado di precisione dell'algoritmo, primo fra tutti la radice dell'errore quadratico medio (*root-mean-square error*, o *RMSE*), così definito:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

In questo caso y_i ed $\hat{y}_i$ rappresentano i valori delle pose stimate e reali, mentre *n* è il numero di pose presenti. Altri valori che vengono misurati

durante i test sono la media, la mediana, la deviazione standard ed i valori di errore minimo e massimo.

I dataset considerati sono spesso molto diversi fra di loro, dovendo valutare la qualità dell'algoritmo in ambienti applicativi differenti. Sono presenti ad esempio dataset indoor che rappresentano una tipica stanza di ufficio, una lunga pavimentazione oppure dataset prettamente di test. Dovendo innanzitutto valutare le differenze fra i tre diversi approcci precedentemente individuati, si è ritenuto utile utilizzare dataset molto diversi fra di loro, in modo da raggiungere una visione generale delle strategie e poter così più facilmente individuare quelle più consone ai singoli casi o in generale, se possibile. In seguito sarà possibile approfondire la soluzione che si riterrà più promettente, arricchendola con risultati provenienti da ulteriori test effettuati sulla stessa.

Oltre ai test di misurazione degli errori per quanto concerne le pose calcolate dall'applicazione, è utile anche elencare i risultati ottenuti in termini di performance. Per misurarli è risultata utile da una parte l'implementazione di un misuratore di FPS e dall'altra l'indicazione del tempo impiegato da alcune fasi dell'algoritmo per eseguire, come ad esempio durante l'estrazione delle features o l'ottimizzazione locale. Questi risultati permettono di confrontare le versioni in termini di performance, ricordando comunque che la fase di ottimizzazione si presenta molto delicata e non è essa stessa variabile fra le diverse strategie di soluzione. La valutazione delle performance si è perciò principalmente concentrata sulla fase di tracking dell'algoritmo, le cui differenze nelle 3 versioni si notano visibilmente. La differenza è spesso evidente anche dal semplice misuratore di FPS disponibile, sebbene esso non offra una misura precisa del tempo di esecuzione.

5.2 Comparazione delle tre configurazioni

La valutazione delle 3 configurazioni è stata impostata utilizzando gli stessi parametri base, per evitare differenze fra esecuzioni diverse dell'applicazione. I parametri degni di nota sono i seguenti:

- *Lunghezza dell'active window*: 5 metri;
- *Soglia di overlapping*: 0.7;
- *Detection di loop avvenuto:* Abilitata;
- *Numero di anelli considerati nell'ottimizzazione*: 3.

Alcuni di questi parametri verranno modificati in seguito per verificare l'andamento dell'algoritmo ed aiutare nella scelta della soluzione ottimale.
I dataset tenuti in considerazione sono i seguenti:

- *fr1/floor*: Ricostruzione del pavimento in legno di una ampia stanza con alcune scrivanie intorno;
- *fr1/desk*: Ricostruzione di una scrivania da ufficio e di una parte delle pareti della stanza;
- *fr1/room*: Ricostruzione di una intera stanza da ufficio di medie dimensioni, con diverse scrivanie ed altri oggetti nei dintorni;
- *fr3/cabinet*: Ricostruzione di una scatola di colore blu a tinta unita, senza altri particolari dettagli;
- *fr3/structure_texture_(near/far)*: Ripresa vicina/lontana di alcuni fogli e strutture di cartone con texture;
- *fr3/nostructure_texture_(far/near_with_loop)*: Ripresa vicina/lontana di alcuni fogli con texture. Nella ripresa vicina viene anche chiuso il loop;
- *fr3/long_office_household*: Ripresa ampia di un ufficio con loop finale.

Di seguito verranno indicati i risultati ottenuti in termini di precisione delle diverse configurazioni, per poi passare all'analisi delle performance.

5.2.1 Valutazione della precisione

La tabella 1.1 mostra i risultati in termini di RMSE, mediana e deviazione standard, riferiti alla strategia ed al dataset utilizzato.

Sequenza		**ORB + ORB**	**ORB + BRISK**	**U-SURF + BRISK**
fr1/floor	RMSE (m)	0.058	0.055	0.051
	median (m)	0.048	0.046	0.046
	std (m)	0.027	0.025	0.019
fr1/desk	RMSE (m)	0.049	0.052	0.042
	median (m)	0.039	0.044	0.028
	std (m)	0.023	0.023	0.024
fr1/room	RMSE (m)	0.270	0.278	0.140
	median (m)	0.251	0.190	0.107
	std (m)	0.103	0.147	0.060
fr3/structure texture near	RMSE (m)	0.092	0.047	0.025
	median (m)	0.090	0.040	0.019
	std (m)	0.021	0.020	0.013
fr3/structure texture far	RMSE (m)	0.052	0.045	0.028
	median (m)	0.038	0.034	0.024
	std (m)	0.025	0.024	0.012
fr3/ nostructure texture near with loop	RMSE (m)	0.046	0.057	0.030
	median (m)	0.039	0.049	0.026
	std (m)	0.019	0.024	0.013
fr3/ nostructure texture far	RMSE (m)	0.178	0.139	0.083
	median (m)	0.082	0.070	0.050
	std (m)	0.145	0.108	0.052
fr3/ long office household	RMSE (m)	0.058	0.063	0.041
	median (m)	0.046	0.052	0.033
	std (m)	0.028	0.028	0.018

Tabella 1.1: Precisione delle soluzioni

Nelle tabelle sono stati sottolineati i valori di RMSE migliori nei diversi dataset, individuando così la soluzione che ottiene il maggior grado di precisione, cioè quella che prevede l'utilizzo del detector U-SURF e dell'extractor BRISK. Il vantaggio è evidentemente causato dal tipo di detection effettuata, la quale risulta molto più affidabile della procedura svolta da ORB, seppur impiegando un tempo di esecuzione maggiore.
Il vantaggio in termini di precisione è poco evidente nei dataset più semplici, come ad esempio nella ricostruzione 3D di una singola scrivania dove il numero di possibili landmarks risulta piuttosto elevato e generalmente affidabile. Le differenze si fanno invece sostanziali nei dataset più ampi, ad esempio *fr1/room* dove con il proseguire della ricostruzione gli errori iniziano ad accumularsi, o in cui i landmarks sono pochi, ad esempio al termine di *fr3/nostructure_texture_far* dove in seguito alla ricostruzione 3D dei fogli la camera si spinge verso il mapping di parte della pavimentazione priva di caratteristiche particolari.
In termini di qualità risulta dunque scontata la superiorità della soluzione U-SURF + BRISK, mentre le altre due soluzioni si presentano tendenzialmente bilanciate: in alcuni casi risulta più vantaggioso l'utilizzo di ORB + BRISK mentre in altri l'opposto. Le differenze sostanziali fra questi due algoritmi verranno sottolineate nella successiva sessione, che presenterà risultati in termini di performance.

5.2.2 Valutazione delle performance

Per valutare le performance delle strategie utilizzate, si è passati alla misurazione del tempo di esecuzione della fase di detection e successiva estrazione delle features. Essendo il numero di keypoints e features variabile in base all'ambiente che si sta esplorando, si è deciso di misurare le performance in base al tempo di esecuzione di detection ed extraction limitando a 500 il numero di features finali utilizzabili, in modo tale da non appesantire l'esecuzione dei successivi moduli dell'algoritmo. Alcuni

algoritmi di detection permettono la limitazione a priori del numero di punti da individuare, mentre per altri ciò non è possibile.
La Tabella 1.2 mostra i risultati ottenuti.

Algoritmi	Detection	Extraction
ORB + ORB	28 ms	72 ms
ORB + BRISK	28 ms	7 ms
U-SURF + BRISK	70 ms	14 ms

Tabella 1.2: Tempo di esecuzione delle soluzioni

Nonostante l'ottimizzazione effettuata sul detector U-SURF, esso si presenta decisamente più lento rispetto al detector ORB. Va notato comunque che tale detector non è limitato nel numero di keypoints da individuare, spesso ottenendo un valore pari a 1000 o superiore che solo di seguito viene troncato per motivi di performance durante le fasi di matching e filtraggio. La detection di un numero elevato di keypoints è garantita in un tempo di esecuzione simile a quello sopra indicato ma permettendo anche di individuare un maggior numero di match di buon livello. Al contrario, ORB è solitamente limitato a priori ad un numero di keypoints da individuare pari a 500: l'aumento di tale numero rallenta in modo maggiore l'algoritmo e non garantisce risultati visibilmente migliori, in quanto la qualità delle features così individuate non è solitamente di un livello superiore alle prime 500. Per tali considerazioni, qualora l'algoritmo volesse considerare un numero di keypoints maggiore, la scelta più conveniente in termini di qualità e performance sarebbe il detector di U-SURF, nel caso invece si voglia mantenere un numero di keypoints non troppo elevato, l'utilizzo del detector di ORB è consigliato. In futuro si potrebbero cercare ulteriori strategie atte a modificare l'algoritmo U-SURF per realizzare una detection limitata nel numero di keypoints, consentendo di ottenere vantaggi in termini di performance.

Per quanto concerne l'operazione di estrazione, l'extractor di ORB si presenta notevolmente più lento, compromettendo la sua usabilità in ambito mobile. Sostituito lo stesso a BRISK e mantenendo l'orientamento dei kepoints per come calcolato dal detector di ORB, si ottengono ottimi risultati in grado di garantire un tempo di esecuzione più vicino a quello desiderato. L'utilizzo di BRISK unito ad U-SURF si presenta invece più rallentato siccome il calcolo dell'orientamento dei keypoints è in questo caso delegato all'extractor di BRISK, il quale dovrà dunque compiere alcune operazioni aggiuntive. Il tempo di esecuzione è comunque piuttosto ridotto e perfettamente accettabile.

In termini più generali, durante l'esecuzione di un dataset come *fr1/desk* l'intero algoritmo, ad esclusione della fase di ottimizzazione, è in grado di eseguire ad una velocità pari a 6-7 FPS nel caso di utilizzo di ORB + ORB oppure di U-SURF + ORB, nel primo caso a causa dei rallentamenti nell'estrazione delle features, nel secondo caso a causa della detection molto dispendiosa. L'utilizzo di ORB + BRISK permette invece di raggiungere velocità fino ai 10 FPS o più.

Unite queste considerazioni ai risultati in termini di precisione, gli algoritmi più promettenti risultano ORB + BRISK ed U-SURF + BRISK, il primo in termini di performance ed il secondo in termini di qualità. Avendo però un magine di miglioramento più elevato nella seconda strategia, siccome le performance si possono migliorare mentre un aumento della precisione comporterebbe un cambio di algoritmo, si predilige questa per le successive considerazioni e come strategia consigliata per l'esecuzione dell'algoritmo SlamDunk su piaffatorma Android.

5.3 Valutazione complessiva dell'algoritmo

Questo paragrafo si soffermerà su una valutazione dell'algoritmo complessivo, aggiungendo considerazioni riguardo alle performance dei moduli fino ad ora tralasciati nel confronto e valutando inoltre i valori che assumono alcuni parametri, precedentemente indicati all'inizio della

Sezione 5.2. Lo scopo di questa analisi è da una parte quello di fornire una visione complessiva dell'andamento del sistema e dall'altra quello di permettere una decisione sui valori più consoni attribuibili ai parametri importanti dell'algoritmo, rendendo note le differenze cruciali quando gli stessi vengono modificati e sottolineando di conseguenza la loro utilità. Infine, verranno mostrati alcuni risultati in termini di ricostruzione 3D.

5.3.1 Tempo di esecuzione

I tempi di esecuzione precedentemente analizzati si riferiscono solo a parte dell'algoritmo, escludendo l'operazione di matching, il filtraggio tramite RANSAC ed infine l'ottimizzazione locale nei casi in cui essa è richiesta. La Tabella 1.3 mostra il range di valori che queste operazioni assumono durante l'esecuzione dell'algoritmo con i parametri della Sezione 5.2.

Operazione	Tempo di esecuzione
Matching	20 ms – 40 ms
RANSAC	1 ms – 10 ms
Ottimizzazione locale	20 ms – alcuni secondi

Tabella 1.3: Tempo di esecuzione matching, RANSAC ed ottimizzazione

La velocità del matching dipende principalmente dal numero di keyframes considerati, pur non variando in maniera considerevole e mantenendosi su valori bassi, il filtraggio tramite RANSAC risulta ancora più rapido, mentre l'ottimizzazione effettuata da g2o presenta maggiori problematiche. Nel caso in cui il numero di nodi e vincoli associati sia basso l'ottimizzazione termina in tempi brevi, mentre all'aumentare soprattutto del numero di vincoli il tempo di esecuzione raggiunge anche alcuni secondi, diventando uno dei maggiori colli di bottiglia del sistema. Anche nel caso di esecuzione in parallelo dell'ottimizzazione non si è in grado di raggiungere risultati considerevolmente migliori, a causa della struttura stessa del problema. Su questa problematica ci si dovrà soffermare in futuro, cercando alternative

efficienti all'ottimizzazione g2o o tentando ulteriori miglioramenti della stessa.

5.3.2 Valutazione dei parametri dell'algoritmo

In questa sezione si analizzeranno i principali parametri utilizzati dall'algoritmo, indicando come le loro modifiche possano variare l'andamento dell'algoritmo:

- *Lunghezza dell'active window*: Determina la dimensione della finestra entro cui prelevare i keyframes. All'aumentare della dimensione si coprirà un'area maggiore ottenendo dei matching più precisi, a scapito di un maggiore utilizzo delle capacità computazionali.
- *Soglia di overlapping*: Un aumento della soglia permette di individuare un nuovo keyframe con variazioni minori rispetto ai keyframes già presenti, ottenendo così una maggiore precisione a causa della ottimizzazione più frequente. Infatti, a parità di traiettoria, aumentando la soglia di overlapping il numero di keyframes risulterà maggiore e di conseguenza maggiore sarà anche il numero di ottimizzazioni effettuate. Come lato negativo vi è il fatto, che aumentando il numero di keyframes e ottimizzazioni, le performance peggiorano di conseguenza. E' importante scegliere questo parametro con criterio siccome un numero troppo alto causerebbe la presenza di keyframes ridondanti, mentre un numero troppo basso comporterebbe un possibile fallimento del tracking o una elevata imprecisione.
- *Detection di loop avvenuto*: Abilitando questa opzione si è in grado di individuare esplicitamente i casi di loop avvenuto ed ottimizzare il grafo di conseguenza. Anche in questo caso, l'attivazione di tale caratteristica aumenterà la precisione della traiettoria, mentre la disattivazione migliorerà le performance.

- *Numero di anelli considerati nell'ottimizzazione*: L'aumento del numero degli anelli spinge verso una ottimizzazione più ampia, con le stesse conseguenze indicate per i due parametri precedenti.

La Tabella 1.4 mostra la variazione, in termini di tempo di esecuzione complessivo e di precisione riferita al RMSE, dei parametri rispetto al caso di default. I dataset usati come riferimento sono *fr1/desk* per quanto riguarda i parametri di loop detection, keyframe overlapping ed il numero di anelli, mentre *fr3/long_office_household* per la variazione dell'active window, la quale richiede una traiettoria più ampia per essere verificata adeguatamente.

Parametro modificato	**Variazione RMSE**	**Variazione tempo di esecuzione**
Detection di loop avvenuto disabilitata	+4%	-6%
Keyframe overlapping = 0.8	-19%	+120%
Keyframe overlapping = 0.6	+53%	-10%
Numero anelli = 5	-9%	+4%
Active window = 7 m	-3%	+5%
Active window = 3 m	+4%	-5%

Tabella 1.4: Valutazione della variazione dei parametri

Dai valori elencati si può notare come il variare del parametro di overlapping dei keyframes non sia consigliabile in entrambe le situazioni, poichè causa peggioramenti da un lato rispetto al tempo di esecuzione e dall'altro rispetto alla precisione raggiunta. Ciò è ancor più evidente qualora si considerasse un dataset più complesso. Gli altri parametri risultano invece più stabili nelle modifiche ed alcuni di essi possono ottenere vantaggi in caso di variazione per dataset semplici, ricordando però che con l'aumento della zona da mappare le modifiche potrebbero non risultare più così vantaggiose. Disabilitare la detection di loop avvenuto non funzionerà

adeguatamente su traiettorie che prevedono un numero maggiore di loop, aumentare il numero di anelli porterebbe nei dataset più complessi ad un tempo di esecuzione insostenibile, mentre la variazione della lunghezza dell'active window appesantirebbe eccessivamente l'applicazione, se troppo grande, oppure compromettere il matching, se troppo piccola.

5.3.3 Ricostruzioni 3D ottenute

Le esecuzioni dell'algoritmo sui dataset precedenti hanno permesso di salvare le pose dei keyframes individuati. Tramite queste pose è possibile ricostruire le point cloud visualizzate dall'applicazione Android su un computer desktop.

Le figure elencate nel seguito mostrano alcuni dei risultati ottenuti.

Figura 1.5: Ricostruzione 3D dataset fr1/floor (visuale dall'alto)

Figura 1.6: Ricostruzione 3D dataset fr1/floor (visuale sul pavimento)

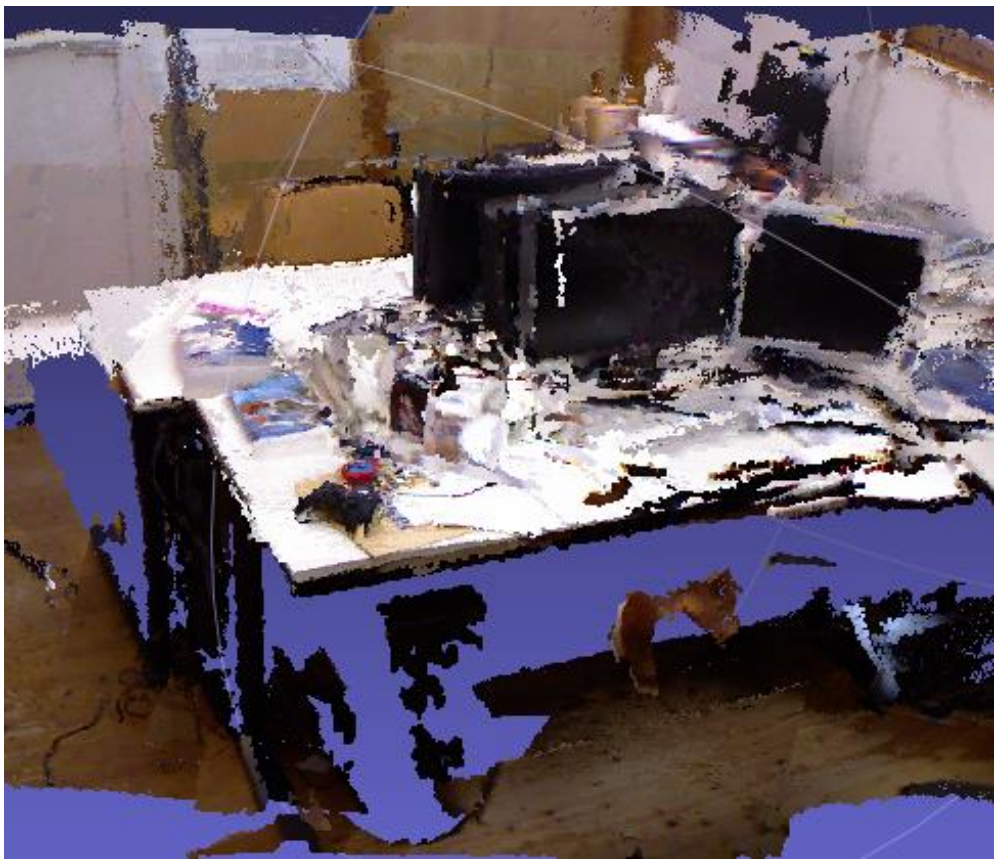

Figura 1.7: Ricostruzione 3D dataset fr1/desk

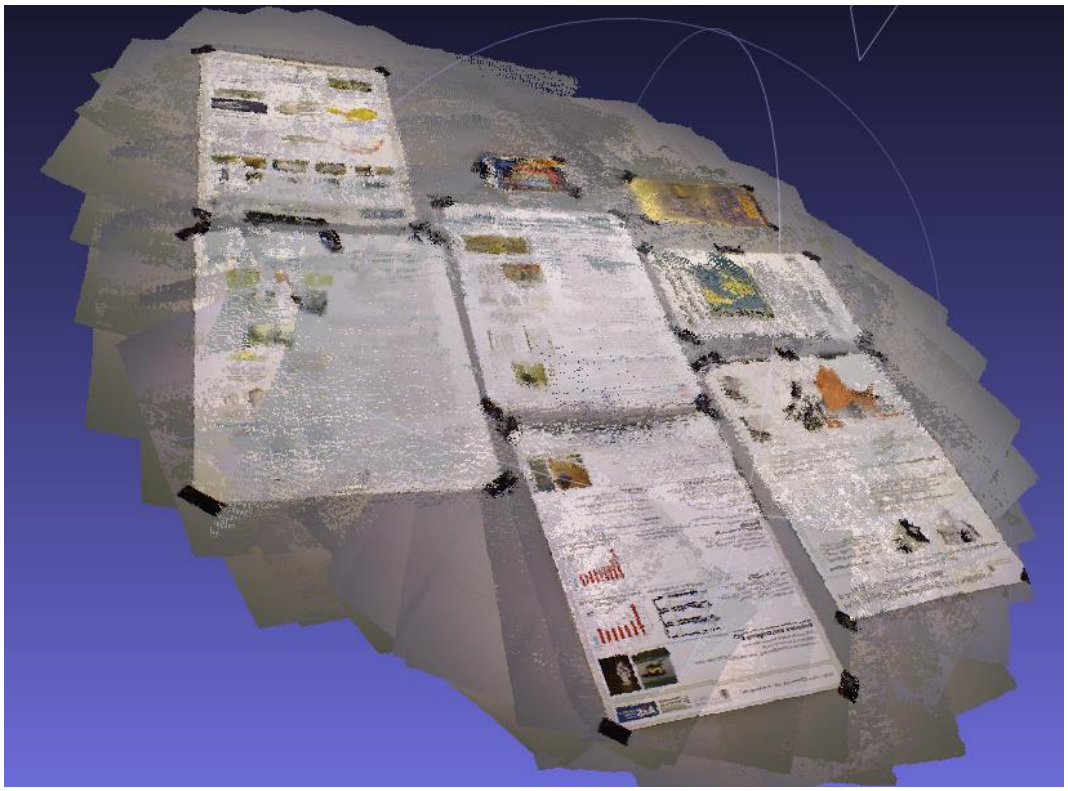

Figura 1.8: Ricostruzione 3D dataset fr3/nostructure_texture_near_with_loop

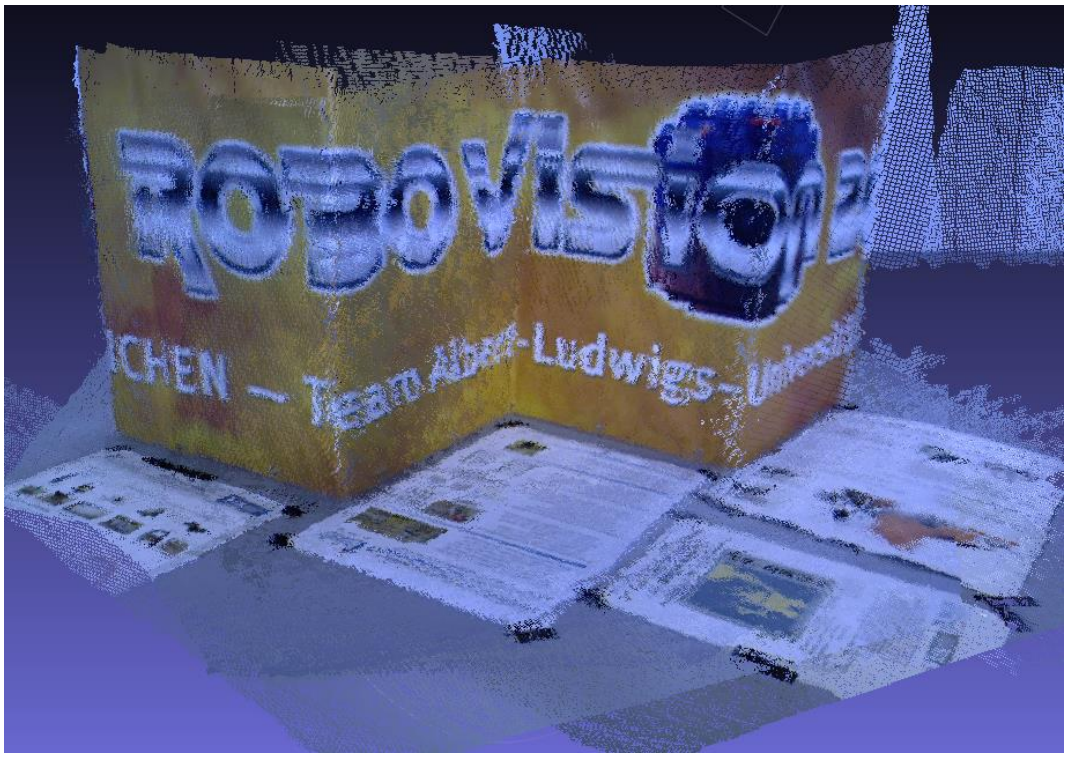

Figura 1.9: Ricostruzione 3D dataset fr3/structure_texture_far

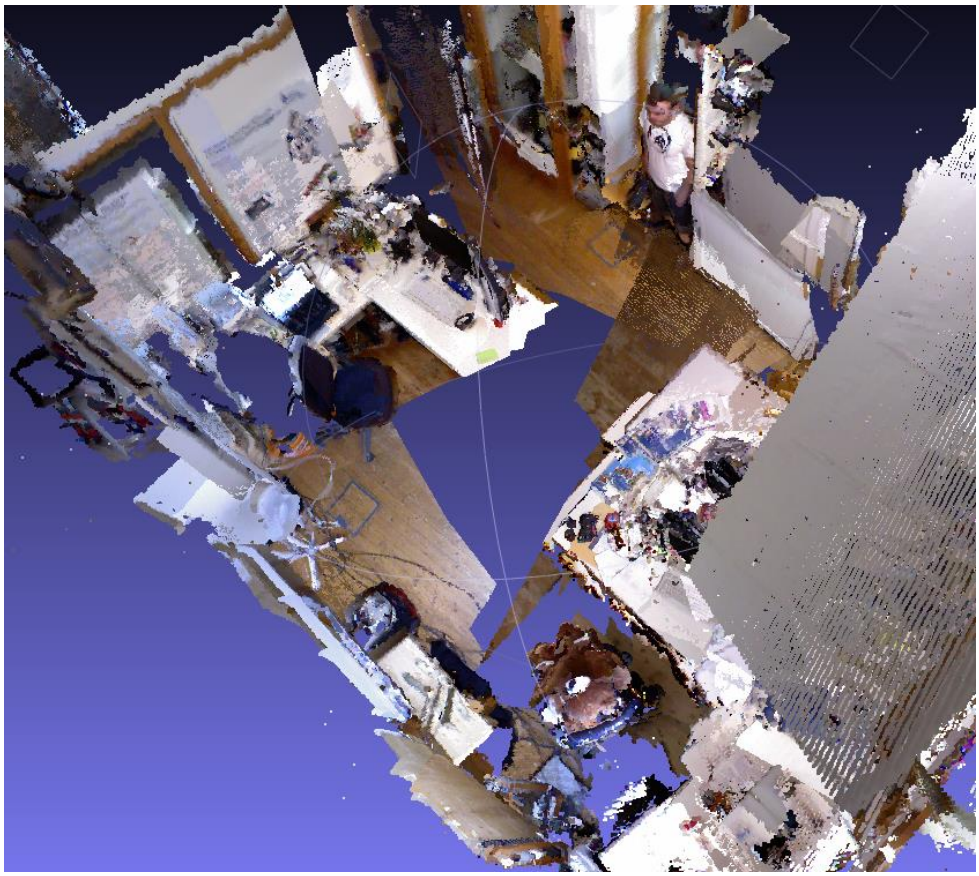

Figura 1.10: Ricostruzione 3D parziale dataset fr1/room

5.4 Utilizzo dell'algoritmo su dispositivi reali

Avendo strutturato l'applicazione in base a componenti dalle funzionalità diverse fra di loro, l'aggiunta di un sensore reale influisce esclusivamente sulla parte riferita al grabber, mentre l'algoritmo in sè e l'output di visualizzazione conseguente non subirà alcuna modifica. Inoltre, le stesse considerazioni effettuate per quanto concerne i dataset valgono anche una volta scelto di utilizzare dispositivi reali, con l'unica accortezza che, contrariamente alla versione desktop di SlamDunk, l'applicazione Android non è attualmente in grado di raggiungere velocità al passo con il frame rate della camera. Per tale motivo si dovrà agire con maggiore attenzione durante il campionamento della zona, evitando un capturing troppo rapido che potrebbe compromettere l'esecuzione corretta dell'algoritmo.

L'applicazione Android è stata eseguita con successo con 3 tipologie diverse di sensori:

- *Asus Xtion Pro Live*: Sensore che supporta la libreria OpenNI per l'interfacciamento, in grado operare a 30 frame al secondo con immagini RGB e depth di risoluzione pari a 640x480, come correntemente utilizzato dall'algoritmo SlamDunk, ed un range di profondità fra 0.8 metri e 3.5 metri;
- *Microsoft Kinect*: Anch'esso supporta la libreria OpenNI, sebbene non nella versione più recente. Di fatto presenta caratteristiche molto simili all'Asus Xtion Pro Live ma ad un range leggermente più elevato;
- *Creative Senz3D*: Webcam 3D recentemente introdotta sul mercato, permette un'acquisizione di immagini RGB a 640x480 e depth a 320x240, con necessità di scalarle alla giusta dimensione, a 30 FPS. Essa inoltre ha a disposizione due modalità di funzionamento, una ravvicinata utile nella ricostruzione di singoli oggetti ed una dal range più elevato per la ricostruzione di ambienti indoor in modo simile ai due sensori precedenti. Non supporta ufficialmente l'utilizzo di OpenNI ma è dotata di driver (attualmente in versione beta) funzionanti su alcuni dispositivi Android, fra cui quello utilizzato nella fase di test.

I primi due sensori si presentano simili fra di loro e di facile utilizzo, ottenendo ottimi risultati anche in ambiente mobile. Il terzo sensore è invece più problematico, a causa dell'utilizzo di driver non ancora in versione definitiva e richiedendo inoltre alcune successive operazioni di manipolazione dell'immagine per ottenere i risultati desiderati. Nonostante ciò, una sua esecuzione su dispositivo mobile risulta promettente e maggiormente adatta agli utilizzi futuri che tali dispositivi dovrebbero avere, sempre più portabili e comodi da agganciare. Ancor più promettente in questo campo risulta il sensore Structure, specificatamente sviluppato per dispositivi mobili ed a breve pronto ad entrare sul mercato.

Le figure di seguito mostano alcune ricostruzioni 3D ottenute tramite l'utilizzo del sensore Asus Xtion Pro Live.

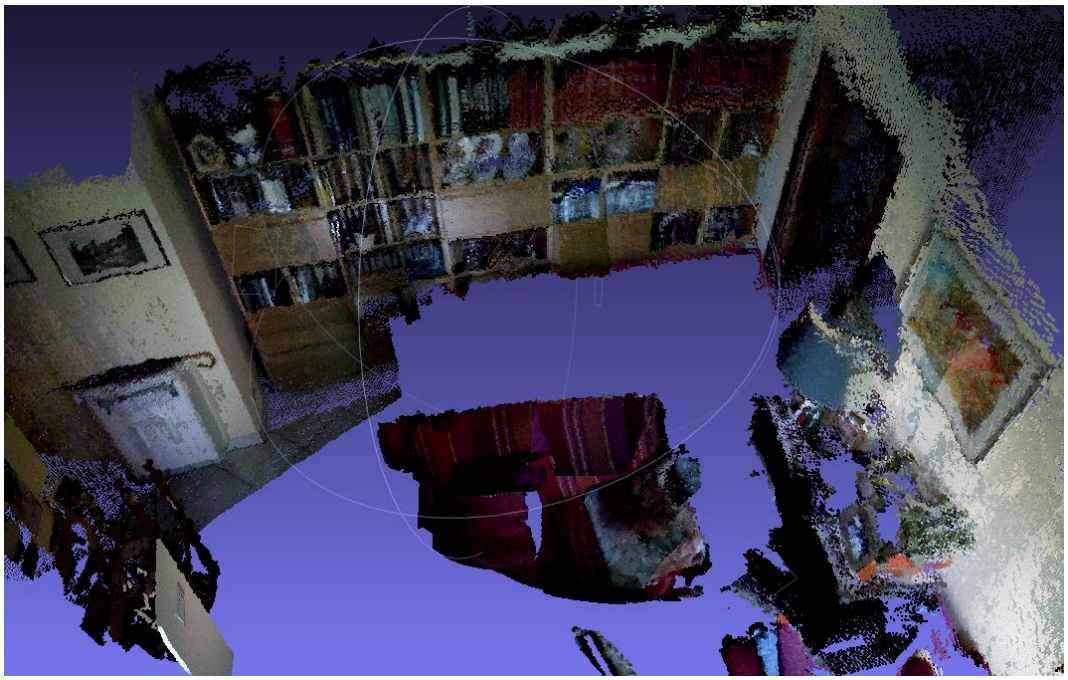

Figura 1.11: Ricostruzione 3D con sensore Xtion Pro Live (1)

Figura 1.12: Ricostruzione 3D con sensore Xtion Pro Live (2)

Considerazioni finali e sviluppi futuri

Questo lavoro di tesi ha comportato la realizzazione del porting su piattaforma mobile Android dell'algoritmo SlamDunk, facendo fronte ad alcuni problemi cruciali di interfacciamento con i sensori disponibili e di ottimizzazione delle performance, allo scopo di rendere fluida l'esecuzione dell'algoritmo su tale ambiente. Sono state adottate diverse strategie di soluzione di cui sono stati successivamente mostrati i risultati sperimentali raggiunti. L'impostazione attuale dell'applicazione Android realizzata fa sì che essa si presenti facilmente estendibile in ogni suo componente ed aperta a successive modifiche e miglioramenti che possano anche comportare l'aggiunta di nuove funzionalità.

In questa ottica è importante sottolineare alcune delle cose che è possibile approfondire per una successiva estensione dell'applicazione, mettendo in primo piano le problematiche di performance. In primo luogo l'applicazione non si presenta per il momento ottimizzata in termini di memoria, se non prevedendo un sampling delle point cloud. Sviluppi futuri potrebbero permettere l'aggiunta di strategie che riducano l'utilizzo della memoria facendo sì che non vi siano più grandi limitazioni sull'esecuzione dell'algoritmo. Un successivo punto di lavoro potrebbe riguardare un miglioramento delle prestazioni dal lato dell'ottimizzazione del grafo, essendo tale operazione molto dispendiosa dal punto di vista computazionale. Si potrebbe perciò pensare di migliorare gli algoritmi adottati durante l'ottimizzazione, adattandoli ad un ambiente mobile, o in alternativa utilizzare metodologie più semplici e rapide, sebbene esse potrebbero comportare un calo in termini di precisione nell'ottimizzazione del grafo. In alternativa, potrebbe essere interessante provare ad utilizzare solutori differenti, ad esempio il CERES solver sviluppato da Google.

Passando invece agli sviluppi futuri meno legati a tematiche di miglioramento delle performance, vi è la possibilità di estendere il numero di sensori 3D con cui l'applicazione è in grado di lavorare, aggiungendo ad esempio il supporto al sensore Structure nonchè a futuri sensori che potrebbero essere disponibili sul mercato, essendo lo stesso in una fase promettente di crescita. Infine, sarebbe utile dotare l'applicazione di funzionalità aggiuntive che permettano di manipolare i dati generati direttamente da dispositivo mobile, rendendola molto più completa ed interessante.

Bibliografia

[AHetal87] K. S. Arun, T. S. Huang, and S. D. Blostein, "Least-Squares Fitting of Two 3-D Point Sets," *Pattern Analysis and Machine Intelligence (PAMI)*, vol. 9, no. 5, pp. 698-700, Sep. 1987.

[BEetal08] D. Borrman, J. Elseberg, K. Lingemann, A. Nuchter, and J. Hertzberg, "Globally Consistent 3D Mapping with Scan Matching," *Journal of Robotics and Autonomous Systems*, vol. 56, pp. 130-142, Feb. 2008.

[BTetal06] H. Bay, T. Tuytelaars, and L. Van Gool, "SURF: Speeded-Up Robust Features," in *European Conference on Computer Vision (ECCV)*, May 2006.

[DRetal07] A. Davison, I. D. Reid, N. D. Molton, and O. Stasse, "MonoSLAM: Real-Time Single Camera SLAM," *Pattern Analysis and Machine Intelligence (PAMI)*, vol. 29, no. 6, pp. 1052-1067, Jun. 2007.

[EHetal12] F. Endres, J. Hess, N. Engelhard, J. Sturm, D. Cremers, and W. Burgard, "An Evaluation of the RGB-D SLAM System," in *International Conference on Robotics and Automation (ICRA)*, May 2012.

[FB81] M. A. Fischler and R. C. Bolles, "Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography," in *Communications of the ACM*, 1981, pp. 381-395.

[FD13] N. Fioraio and L. Di Stefano, "SlamDunk: Affordable Real-Time RGB-D SLAM," 2013.

[G00] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[GB10] G. Guennebaud and B. Jacob, "Eigen v3," 2010.

[K10] K. Group, "OpenGL ES Common Profile Specification Version 2.0.25," Nov. 2010.

[KGetal11] R. Kummerle, G. Grisetti, H. Strasdat, K. Konolige, and W. Burgard, "g2o: A General Framework for Graph Optimization," in *International Conference on Robotics and Automation (ICRA)*, May 2011.

[KM07] G. Klein and D. Murray, "Parallel Tracking and Mapping for Small AR Workspaces," in *International Symposium on Mixed and Augmented Reality (ISMAR)*, Nov. 2007, pp. 225-234.

[L04] D. G. Lowe, "Distinctive Image Features from Scale-Invariant Keypoints," *International Journal of Computer Vision*, pp. 91-110, 2004.

[Link01] C. Strachniss. (2012) Robot Mapping: EKF SLAM. [Online]. http://ais.informatik.uni-freiburg.de/teaching/ws12/mapping/pdf/slam04-ekf-slam.pdf

[Link02] C. Strachniss. (2012) Robot Mapping: Hierarchical Pose-Graphs for Online Mapping. [Online]. http://ais.informatik.uni-freiburg.de/teaching/ws12/mapping/pdf/slam16-hierarchical.pdf

[Link03] C. Strachniss. (2012) Robot Mapping: Graph-Based SLAM with Landmarks. [Online]. http://ais.informatik.uni-freiburg.de/teaching/ws12/mapping/pdf/slam17-ls-landmarks.pdf

[Link04] Structure Sensor WebSite. [Online]. http://structure.io/

[Link05] Android Developer WebSite. [Online]. https://developer.android.com

[LJ07] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li, "Multi-Probe LSH: Efficient Indexing for High-Dimensional Similarity Search," in *International Conference on Very Large Data Bases*, 2007.

[LSetal11] S. Leutenegger, M. Chli, and R. Y. Siegwart, "BRISK: Binary Robust Invariant Scalable Keypoints," in *International Conference on Computer Vision (ICCV)*, 2011.

[ML09] M. Muja and D. G. Lowe, "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration," in *VISAPP*, 2009.

[ML12] M. Muja and D. G. Lowe, "Fast Matching of Binary Features," in *Conference on Computer and Robot Vision (CRV)*, 2012.

[MT04] M. Montemerlo and S. Thrun, "Large-Scale Robotic 3-D Mapping of Urban Structures," in *International Symposium on Experimental Robotics (ISER)*, 2004.

[NLetal11] R. Newcombe, S. Lovegrove, and A. Davison, "DTAM: Dense Tracking and Mapping in Real-Time," in *International Conference on Computer Vision (ICCV)*, 2011.

[O11] OpenNI, "The Open Natural Interaction Framework," Jan. 2011.

[RC11] R. B. Rusu and S. Cousins, "3D is here: Point Cloud Library (PCL)," in *International Conference on Robotics and Automation (ICRA)*, May 2011.

[RRetal11] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "ORB: An Efficient Alternative to SIFT or SURF," in *International Conference on Computer Vision (ICCV)*, 2011.

[SDetal11] H. Strasdat, A. J. Davison, J. Montiel, and K. Konolige, "Double Window Optimisation for Constant Time Visual SLAM," in *International Conference on Computer Vision (ICCV)*, 2011, pp. 2352-2359.

[SEetal12] J. Sturm, N. Engelhard, F. Endres, W. Burgard, and D. Cremers, "A Benchmark for the Evaluation of RGB-D SLAM Systems," in *International Conference on Intelligent Robots and Systems (IROS)*, 2012.

[SMetal09] G. Sibley, C. Mei, I. Reid, and P. Newman, "Adaptive Relative Bundle Adjustment," in *Robotics: Science and Systems (RSS)*, 2009.