ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

Corso di laurea in Ingegneria Informatica

LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

TESI DI LAUREA

in

Computer Vision & Image Processing M

REALIZZAZIONE DI UN SISTEMA DI RICOSTRUZIONE 3D SU PIATTAFORMA MOBILE

CANDIDATO:
Mirko Collura

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

CORRELATORI:
Dott. Ing. **Nicholas Brunetto**
Dott. Ing. **Tommaso Cavallari**

Anno Accademico 2014/2015

Sessione III

Ringraziamenti

Al termine di questo lungo e impegnativo percorso di studi un ringraziamento è sentito e dovuto a tutte le persone che nel loro piccolo o con grande presenza mi hanno accompagnato verso questo traguardo.

In primo luogo penso di dover ringraziare i miei genitori che hanno sempre creduto in me e mi hanno sostenuto in ogni scelta, in particolare quella Universitaria, sia moralmente che economicamente facendo grandi rinunce e sacrifici, permettendomi di raggiungere questo traguardo senza farmi mai mancare nulla.

Un ringraziamento particolare va ad una persona Speciale, la mia ragazza, *Eleonora*, per avermi spronato, incoraggiato ed essere riuscita sempre a curare (e sopportare) i miei lunghissimi momenti d'ansia preesame accompagnandomi ad ogni appello. La ringrazio per essermi stata vicino anche nei momenti più difficili, sostenendomi e credendo sempre in me. Grazie!

Un grazie anche ai miei colleghi, ma soprattutto amici, *Francesco* e *Annalucia* con i quali ho condiviso un'innumerevole ore di studio e che con i loro aiuti e consigli hanno reso sicuramente questo percorso di studi più semplice e piacevole. Li ringrazio per le passeggiate spensierate in centro sorseggiando la solita birretta. ☺

Un grazie speciale va ad un Grande Amico e collega, *Costantino*, con il quale abbiamo condiviso lo studio della maggior parte delle materie di questo percorso universitario; in merito a questo mi scuso per tutte le volte che l'ho svegliato presentandomi a casa sua alle 8.30 del mattino (per iniziare la giornata di studio) suonando senza pietà il citofono. Lo ringrazio per aver condiviso anche le attività extrauniversitarie, come la palestra, il calcetto del venerdì e gli innumerevoli ma soprattutto "obesizzanti" pranzi e cene.

Per la sua grande chiarezza e disponibilità e per avermi dato l'opportunità di lavorare su questo argomento di tesi, ringrazio il Prof. *Luigi Di Stefano.*

Un grazie è dovuto anche ai correlatori di questa tesi: *Ing. Nicholas Brunetto* e *Ing. Tommaso Cavallari,* per avermi accolto sin da subito nel piacevolissimo ambiente del laboratorio di Computer Vision e avermi seguito in maniera attenta, professionale e puntigliosa fino alla conclusione di questo elaborato.

Sommario

Indice delle figure

Introduzione

Negli ultimi anni, una delle tematiche chiave nella Computer Vision è stata la possibilità di ottenere ricostruzioni 3D di piccoli oggetti o grandi scene, tramite l'utilizzo di particolari sensori, come laser scanner e sensori RGB-D[1]. A tal proposito sono stati sviluppati diversi algoritmi e la ricerca su questo campo è in continua e rapida evoluzione.

Molti degli algoritmi sviluppati, come ad esempio l'algoritmo *KinectFusion* a cui faremo riferimento in questo lavoro, per essere eseguiti necessitano di un hardware molto performante. Per tale motivo, essi sono stati per lo più eseguiti su sistemi PC desktop o al massimo su alcuni computer portatili di fascia alta.

La recente espansione del mondo mobile ha però incentivato la curiosità di ottenere delle versioni di tali algoritmi sui nuovi e più pratici dispositivi. Inoltre, l'affermarsi di sensori più piccoli, leggeri e portatili (come ad esempio il sensore *Structure* della Occipital) ha incentivato ulteriormente la ricerca in questa direzione.

Questo testo mira a descrivere le metodologie, tecnologie e strumenti, sia hardware che software, messi in atto nel processo di porting dell'algoritmo *KinectFusion*, fino ad ora utilizzato solo su sistemi operativi Linux e Windows su PC desktop, ad Android, piattaforma ormai molto diffusa per i dispositivi mobili. Nello specifico, come dispositivo di sviluppo è stato scelto il tablet *Shield* della NVIDIA. Definiremo più avanti le motivazioni di questa scelta.

L'elaborato che verrà presentato è stato svolto durante l'attività di tesi di laurea magistrale in Ingegneria Informatica presso il laboratorio di Computer Vision

[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.

della facoltà di Ingegneria di Bologna. Vediamo ora un dettaglio sugli argomenti che tratterà questo lavoro di tesi.

Nel primo capitolo si farà una panoramica generale sull'algoritmo KinectFusion, valutandone le peculiarità e le varie fasi di cui è composto. Esamineremo quindi il processo che va dall'acquisizione dell'input dell'algoritmo (immagine depth), fino all'estrazione dell'immagine di ricostruzione della scena 3D.

Nel secondo capitolo si passerà alla descrizione del linguaggio CUDA, elemento chiave per l'esecuzione efficiente dell'algoritmo KinectFusion. Si illustreranno le tecnologie hardware e software che questo framework comprende e si mostrerà il suo impiego all'interno dell'implementazione KinectFusion di riferimento. Al termine del capitolo, faremo quindi una veloce panoramica sull'hardware utilizzato per il porting dell'algoritmo su piattaforma Android, ovvero il tablet Shield.

Il terzo capitolo riguarda le tecniche di deployment e gli strumenti software utilizzati per la compilazione il codice dell'algoritmo sul nuovo sistema, nonché dei vari moduli, librerie e dipendenze che l'attuale implementazione di KinectFusion su dispositivi mobili sfrutta per poter eseguire.

Il quarto capitolo mostrerà le varie caratteristiche dell'applicazione mobile sviluppata. Verranno descritti sia gli aspetti grafici, per quanto concerne l'interfaccia e le interazioni con l'utente, sia come sono stati gestiti i comportamenti dei vari elementi dell'applicazione, al fine di rendere il tutto coerente e fruibile anche ad una utenza poco esperta nel campo della ricostruzione 3D.

Nel quinto capitolo, andremo a valutare i risultati ottenuti, facendo prima una valutazione qualitativa delle ricostruzioni e successivamente una valutazione prestazionale in termini di tempo di esecuzione al variare di alcuni parametri di configurazione dell'algoritmo KinectFusion.

Infine, al termine del lavoro verranno elencate le conclusioni ed i possibili sviluppi futuri dell'elaborato.

Capitolo 1
L’algoritmo KinectFusion

La ricostruzione 3D di piccoli oggetti o di più grandi scene è stata negli ultimi tempi un tema di ricerca molto diffuso nell’ambito della Computer Vision. Nel corso degli ultimi anni, infatti, numerosi algoritmi di ricostruzione tridimensionale sono stati proposti in letteratura. Uno tra i più famosi è KinectFusion [1], il quale riesce ad ottenere in modo efficiente una ricostruzione tridimensionale della scena, sfruttando diversi algoritmi di Computer Vision e rappresentando la scena stessa all’interno di un volume virtuale. Questo volume è a sua volta composto da numerosi elementi chiamati *voxel*, ognuno contenente determinati valori ed occupante una posizione ben precisa all’interno del volume (analogo al pixel ma all’interno di una immagine tridimensionale). Grazie a tale struttura e al fatto che i dati in essa contenuti, come vedremo, sono in continuo aggiornamento durante le varie iterazioni dell’algoritmo, KinectFusion è in grado di correggere eventuali imprecisioni della scena ricostruita.

KinectFusion è un algoritmo iterativo che include al suo interno numerose fasi. Ognuna di queste fasi può essere implementata utilizzando metodologie differenti, scelte in base allo scenario d’uso ed alle performance desiderate. L’algoritmo KinectFusion riesce a ricreare un modello 3D della scena a partire da dati in input sotto forma di immagini depth. Tali immagini vengono acquisite da un apposito sensore di cui KinectFusion ne calcola ad ogni iterazione la posizione e l’orientamento (posa) corrente. Una conseguente operazione di integrazione permette di riempire opportunamente la struttura dati volumetrica presentata precedentemente ed infine, mediante un’operazione di raycasting, navigando all’interno del volume si ottiene una vista della ricostruzione attuale.

1.1 Input dell'algoritmo

Come precedentemente specificato, l'input tipico dell'algoritmo KinectFusion è rappresentato da una immagine depth (anche chiamata *depth map*), nella quale il contenuto informativo dei pixel non rappresenta il colore reale dei punti catturati dalla telecamera, come nelle immagini RGB acquisite dalle comuni fotocamere, ma piuttosto la distanza dei vari punti appartenenti alla scena 3D dalla telecamera/sensore. A tale input è anche possibile abbinare una immagine RGB che sarà utile all'atto di visualizzazione del modello 3D risultante ma non influente nell'esecuzione dell'algoritmo complessivo, almeno nella versione a cui si fa riferimento.

Il sensore che è stato utilizzato per la maggior parte delle esecuzioni di KinectFusion su PC desktop è il sensore *Kinect* della Microsoft, visibile in **Figura 1**, mentre quello adoperato per lo sviluppo di questa tesi in ambito mobile è il sensore *Structure* della Occipital, visibile in **Figura 2**.

Figura 1: Sensore Kinect

Figura 2: Sensore Structure

Entrambi i dispositivi sono equipaggiati con un sensore a infrarossi tramite il quale riescono ad acquisire le immagini di depth che servono come input

all’algoritmo. Il Kinect, dal cui nome deriva appunto il nome dell’algoritmo stesso, rispetto allo Structure ha in più la normale fotocamera a colori per l’acquisizione delle immagini (RGB). Lo Structure ha come vantaggio la caratteristica di essere più piccolo, portatile e comodamente adattabile ai dispositivi mobili, di cui ne può sfruttare la fotocamera RGB in caso si volesse aggiungere il colore alle ricostruzioni 3D.

La **Figura 3** rappresenta una scena indoor, estratta da un noto dataset [3], ed è un esempio di immagine depth. La rappresentazione della distanza in questa immagine, è stata mappata sulla scala di grigi. Più precisamente, a punti più vicini al sensore sono assegnati colori più scuri, a punti più lontani colori più chiari. Come possiamo notare, infatti, i punti della scrivania e dei monitor dei pc risultano più scuri della persona e della parete sullo sfondo, essendo questi ultimi più distanti dal sensore utilizzato.

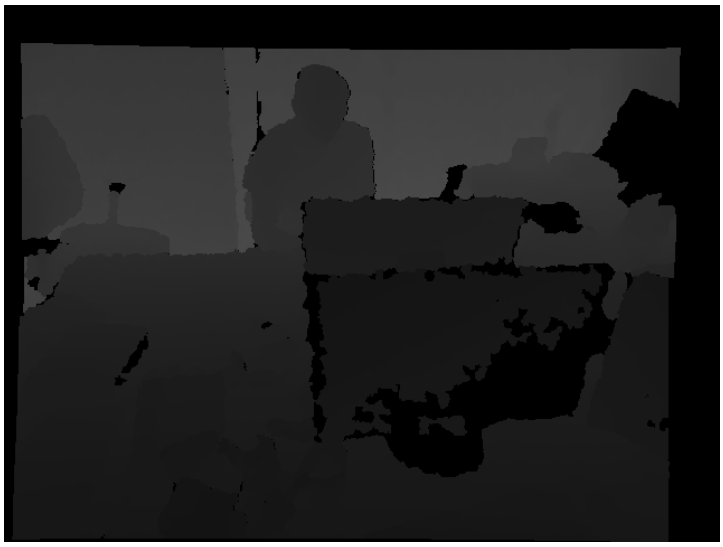

Figura 3: Esempio di immagine depth

1.2 Le fasi dell’algoritmo

Abbiamo detto, quindi, che l’algoritmo KinectFusion è composto da diverse operazioni che si ripetono in sequenza. Ad ogni i-esima iterazione i passaggi sono i seguenti:

- **Conversione della depth map:** Viene convertita l'i-esima immagine depth in una mappa 3D di vertici nel sistema di riferimento solidale con la camera e viene effettuato il calcolo dei vettori delle normali ai punti.
- **Camera Tracking:** Viene eseguito il tracking della posa della telecamera, ovvero si cerca di ottenere la matrice di trasformazione T_i che permette di passare dal sistema di riferimento della camera al sistema di riferimento globale. Il tracking può essere effettuato utilizzando diversi algoritmi, fra cui il tracking geometrico o sfruttando il diffuso algoritmo *Iterative Closest Point* (ICP).
- **Integrazione volumetrica:** In seguito è effettuata l'integrazione volumetrica del modello, nella quale le coordinate voxel vengono mappate in coordinate globali e ad ogni voxel viene associato un valore in base ad una *Truncated Signed Distance Function* (TDSF), il quale viene successivamente mediato con i valori delle precedenti iterazioni.
- **Raycasting:** Infine viene fatto il raycasting del volume per fornire all'utente una visione esplicita della superficie ricostruita. Il raycasting fornisce inoltre una depth map sintetica che può essere utilizzata per migliorare il tracking ICP all'iterazione successiva.

Per ognuno di questi passaggi, sono stati sviluppati algoritmi efficienti basati su implementazioni GPU in linguaggio CUDA. Tali implementazioni parallelizzano l'esecuzione del codice permettendo all'algoritmo di essere eseguito in real-time. I vantaggi prestazionali dell'esecuzione del codice in parallelo, il linguaggio CUDA e la sua implementazione su GPU saranno spiegati nel capitolo successivo.

Vedremo ora un po' più nel dettaglio i passaggi dell'algoritmo KinectFusion [1].

1.2.1 Conversione della depth map

Al fine di sostenere il passo successivo di tracking della camera i punti nell'immagine depth devono essere convertiti in uno spazio 3D, in un primo momento nel sistema di riferimento della telecamera e, per i passaggi successivi dell'algoritmo, in quello globale.

Prima di procedere con il mapping tra le coordinate, l'immagine depth viene sottoposta ad un filtro bilaterale, il quale rimuove l'eventuale rumore dall'immagine senza attenuare però i contenuti semantici significativi (edge). Questa operazione è utile soprattutto ai fini del calcolo dei vettori delle normali ai punti in quanto esso, nell'implementazione di riferimento, per ogni punto dell'immagine 2D è funzione del punto stesso e dei sui vicini, quindi una variazione brusca di intensità tra pixel vicini dovuta al rumore porterebbe a vettori delle normali errati.

Una volta ottenuta l'immagine di depth filtrata si considera la matrice *K* degli intrinseci della telecamera per effettuare la trasformazione dei punti nello spazio 3D. La matrice *K* si ottiene attraverso la calibrazione del sensore utilizzato e viene definita come:

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

dove 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. E' possibile dunque passare dallo spazio 2D dell'immagine di depth allo spazio 3D nel sistema di riferimento della telecamera, tramite la seguente operazione:

$$v_i(x, y) = D_i(x, y) \cdot K^{-1} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix},$$

dove *x* ed *y* rappresentano le coordinate di un generico pixel nell'immagine depth, $D_i(x, y)$ è la misura di profondità misurata tramite le coordinate specificate e K^{-1} è l'inversa della matrice degli intrinseci. Risulta inoltre utile indicare l'indice *i*, il quale rappresenta l'iterazione corrente dell'algoritmo ed è di aiuto nel discriminare i vertici e le normali attuali da quelli ottenuti nell'iterazione precedente.

Una volta che si è eseguita in parallelo questa operazione per tutte le coordinate di pixel dell'immagine 2D, si ottiene la mappa dei vertici V_i nello spazio 3D nel sistema di riferimento della telecamera.

I vettori normali corrispondenti ad ogni vertice sono invece calcolati sui punti riproiettati che erano vicini nell'immagine 2D, tramite:

$$n_i(x,y) = \big(v_i(x+1,y) - v_i(x,y)\big) \times \big(v_i(x,y+1) - v_i(x,y)\big),$$

dove l'operatore × rappresenta il prodotto vettoriale. Questo fornisce una mappa delle normali N_i, anch'essa calcolata in parallelo.

1.2.2 Camera Tracking: l'algoritmo ICP

Il tracking spaziale della posizione della telecamera rispetto al sistema di riferimento globale è un'operazione di fondamentale importanza nell'algoritmo KinectFusion. Esso consiste nella stima ad ogni i-esima iterazione della matrice di trasformazione rigida

$$\mathrm{T_i} = \begin{bmatrix} \mathrm{R_i} & \mathrm{t_i} \\ \overline{0}_3^{\mathrm{T}} & 1 \end{bmatrix},$$

il che corrisponde a conoscere la posa nello spazio 3D della telecamera rispetto al sistema di riferimento globale. Nella definizione R_i rappresenta la matrice 3x3 di rotazione, mentre t_i è il vettore di traslazione. Una volta conosciuta questa matrice è possibile operare la trasformazione fra il sistema di riferimento della telecamera ed il sistema di riferimeno globale, nel seguente modo per le coordinate dei punti:

$$\begin{bmatrix} \mathrm{v_i^g(x,y)} \\ 1 \end{bmatrix} = \mathrm{T_i} \cdot \begin{bmatrix} \mathrm{v_i(x,y)} \\ 1 \end{bmatrix},$$

mentre per i vettottori normali l'operazione è la seguente:

$$n_i^g(x,y) = R_i \cdot n_i(x,y).$$

E' anche possibile effettuare l'operazione inversa, come si vedrà nel seguito.

Uno dei metodi più diffusi che è in grado di stimare la matrice di trasformazione è quello dell'algoritmo *Iterative Closest Point* (ICP), il quale genera ad ogni frame una stima della matrice di rototraslazione che in maniera migliore allinea la scena catturata all'iterazione precedente con quella attuale.

Il primo passaggio dell'algoritmo ICP riguarda la ricerca dei punti corrispondenti fra le due rappresentazioni della scena. Nell'implementazione a cui facciamo riferimento questa ricerca viene effettuata tramite una metodologia chiamata *projective data association*. Questo metodo differisce da quello classico in quanto considera una riproiezione dei punti 3D dell'iterazione precedente sul

piano immagine per trovare le corrispondenze con i punti 3D attuali, mentre comunemente la ricerca dei vicini avverrebbe soltanto considerando le distanze fra i punti dei due frame nello spazio 3D.

Lo pseudo-codice seguente riassume i passaggi della procedura di calcolo dei punti corrispondenti:

$for\ each\ (x, y) \in \mathbf{D_i}$ in parallel do

  if $\mathbf{D_i}(\mathbf{x}, \mathbf{y}) > \mathbf{0}$ **then**

    $\begin{bmatrix} v_{i-1}(x,y) \\ 1 \end{bmatrix} = T_{i-1}^{-1} \cdot \begin{bmatrix} v_{i-1}^{g}(x,y) \\ 1 \end{bmatrix}$

    $p = (\bar{x}, \bar{y}) \leftarrow$ perspective project vertex v_{i-1}

    if $\mathbf{p} \in \mathbf{V_i}$ **then**

      $\begin{bmatrix} v \\ 1 \end{bmatrix} = T_i \cdot \begin{bmatrix} v_i(p) \\ 1 \end{bmatrix}$

      $n = R_i \cdot n_i(p)$

      if $\|v - v_{i-1}^{g}\| <$ distance threshold **and**

      $\text{abs}(n \cdot n_{i-1}^{g}) <$ normal threshold **then**

        point correspondence found

Per ogni coordinata *(x, y)* nell'immagine di depth corrente, se il rispettivo valore è significativo (maggiore di 0), si trasformano i vertici 3D dell'iterazione precedente dalle coordinate 3D globali allo spazio 3D della telecamera, applicando la matrice inversa T_{i-1}^{-1} di trasformazione rigida (tra i due sistemi di riferimento) del frame precedente. Successivamente si proiettano sul piano immagine i v_{i-1} ottenuti al passo precedente ottenendo i punti 2D *p*. Se un determinato *p* ha un corrispettivo nell'attuale mappa dei vertici V_i allora tale punto viene portato nel sistema di riferimento globale tramite la trasformazione T_i (inizialmente inizializzata a T_{i-1}). Anche il vettore delle normali viene trasformato tra i due sistemi di riferimento. Infine, se la distanza euclidea e l'angolo tra le normali dei due punti rispettano una certa soglia, allora la corrispondenza è stata trovata.

Dati i punti corrispondenti nei due frame, i criteri in base ai quali si valuta l'allineamento fra gli stessi variano in base all'implementazione di ICP utilizzata. In questo caso ci concentreremo sull'implementazione che minimizza la distanza cosiddetta *point-to-plane*, la quale non rappresenta la semplice distanza euclidea fra punti corrispondenti nei due frame, bensì la distanza tra ogni punto nel frame corrente e il piano tangente ai punti corrispondenti del frame precedente.
La matrice di trasformazione rigida T_i è perciò calcolata come la trasformazione che minimizza un errore geometrico tra i punti del piano, tramite la seguente formula:

$$\arg\min \sum_{\substack{u \\ D_i(u)>0}} ||(R_i \cdot v_i(u) + t_i - v^g_{i-1}(u)) \cdot n^g_{i-1}(u)||^2$$

In questo caso la trasformazione è stata scomposta rispettivamente nella traslazione t_i e nella rotazione R_i per facilitare la comprensione.
ICP è un algoritmo iterativo, perciò la procedura prosegue nuovamente con la ricerca dei punti corrispondenti utilizzando la matrice di trasformazione appena ricavata e minimizzando di conseguenza il nuovo errore calcolato. L'operazione prosegue fino alla convergenza, la quale può essere misurata considerando una soglia sotto la quale l'errore è ritenuto accettabile e l'algoritmo può quindi terminare con successo.

1.2.3 Integrazione volumetrica e TSDF

Per ricostruire le superfici tridimensionali KinectFusion sfrutta una rappresentazione volumetrica dello spazio. Viene istanziato in memoria un volume 3D di risoluzione variabile *X*, i valori ammessi sono quelli per cui *X* % 32 = 0. Sebbene questa rappresentazione sia molto dispendiosa in termini di memoria, il fatto che il volume sia allocato sulla GPU permette un accesso condiviso dei voxel da parte dell'algoritmo di integrazione volumetrica, favorendo dei tempi di integrazione real-time.
L'integrazione volumetrica può essere effettuata seguendo lo schema del seguente pseudo-codice:

for each *voxel g in* (x, y) volume slice **in parallel do**

while sweeping from front slice to back **do**

$v^g \leftarrow$ convert g from grid to global 3D position

$$\begin{bmatrix} v \\ 1 \end{bmatrix} = T_i^{-1} \cdot \begin{bmatrix} v^g \\ 1 \end{bmatrix}$$

$p = (\bar{x}, \bar{y}) \leftarrow$ perspective project vertex v

if v in camera view frustum **then**

$sdf_i = D_i(p) - \|t_i - v^g\|$

if $(sdf_i > 0)$ **then**

$tsdf_i = \min(1,\ sdf_i / \max \text{truncation})$

else

$tsdf_i = \max(-1,\ sdf_i / \min \text{truncation})$

$w_i = \min(\max \text{ weight},\ w_{i-1} + 1)$

$$tsdf^{avg} = \frac{tsdf_{i-1} \cdot w_{i-1} + tsdf_i \cdot w_i}{w_{i-1} + w_i}$$

store w_i and $tsdf^{avg}$ at voxel g

Essendo il numero di voxel molto elevato (per esempio $512^3 \approx 134$ milioni di voxels) è impossibile assegnare ad ognuno di essi un thread GPU, quindi sono assegnati solo i voxel in tutte le posizioni *(x, y)* della faccia frontale e successivamente viene fatta scorrere la finestra lungo l'asse *z* del volume, come in **Figura 4**.

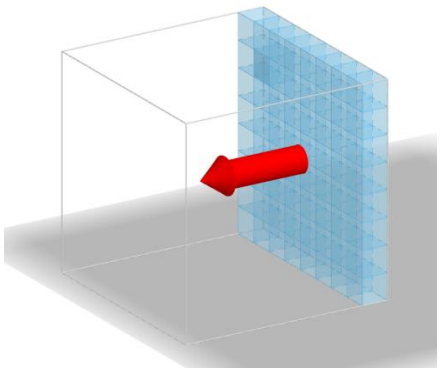

Figura 4: Esempio di voxel di una faccia assegnati ai thread e direzione di scorrimento della finestra

Quindi per ogni voxel della finestra, vengono convertite le coordinate voxel in coordinate mondo, secondo la scala di distanza metrica che si vuole rappresentare. Infatti, è possibile con lo stesso numero di voxel rappresentare diverse misure di distanze in metri, come è possibile vendere nelle **Figure 5** e **6**.

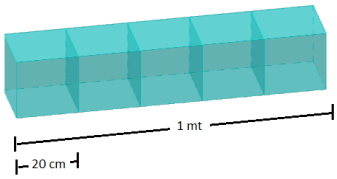

Figura 5: Lato da 1 metro

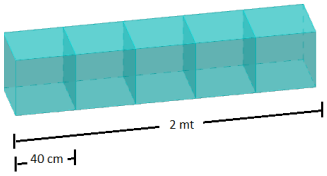

Figura 6: Lato da 2 metri

Per esempio, se si volesse rappresentare una distanza di 1 metro con 5 voxel allora ognuno di essi rappresenterebbe un range di 20 cm, invece, se la distanza che si volesse riprodurre fosse di 2 metri, ogni voxel rappresenterebbe 40 cm. Queste misure, sono soltanto a scopo illustrativo del rapporto:

$$\text{dimensione voxel} = \frac{\text{dimensione lato}}{\text{numero di voxel per lato}}.$$

Nelle applicazioni reali, infatti, se si utilizza una rappresentazione volumetrica da 512 voxel per lato e si volesse rappresentare 2 metri di scena, allora ogni voxel rappresenterebbe un range di distanze di 200 cm / 512 voxel = 0,39 cm/voxel. E' facile intuire, quindi, che un aumento del numero di voxel per lato (a parità di distanza ricoperta), o una diminuzione della misura della scena che si vuole rappresentare (a parità di voxel per lato), comporterebbe un miglioramento nella qualità della ricostruzione. E' importante sottolineare che il volume di ricostruzione non è necessariamente un cubo, ma può riportare misure diverse lungo le tre dimensioni, da questo consegue una possibile differenza di risoluzione nelle varie dimensioni.

Una volta trasformate le coordinate voxel in coordinate globali v^g, tali coordinate subiscono un'ulteriore trasformazione nel sistema di riferimento telecamera v e successivamente in quello immagine 2D p tramite proiezione prospettica.

Se *v* è all'interno del frustum della camera, allora viene calcolata la *Signed Distance Function* (SDF) come:

$$\mathrm{sdf_i} = \mathrm{D}_{\mathrm{i}}(\mathrm{p}) - \|\mathrm{t_i} - \mathrm{v^g}\|$$

cioè una funzione con segno rappresentante la differenza tra la distanza di depth del *p*-esimo punto registrata nell'attuale iterazione, con la distanza dalla telecamera della posizione del voxel v^g in coordinate globali (tramite sottrazione del vettore di traslazione). Ad ogni iterazione una nuova misura di SDF è calcolata per ogni voxel correntemente visibile dal sensore.

Se SDF_i risulta essere un valore maggiore di 0, vuol dire che la posizione del voxel in esame, nel sistema di riferimento globale, si trova davanti la reale superficie, altrimenti si troverà dietro. L'SDF viene inoltre troncata oltre una certa distanza, creando una *Truncated Signed Distance Function* (TSDF), la quale è salvata all'interno di ogni voxel, mediata con dei pesi ed aggiornata ad ogni iterazione. In **Figura 7** è possibile vedere un esempio 2D di questa funzione di distanza.

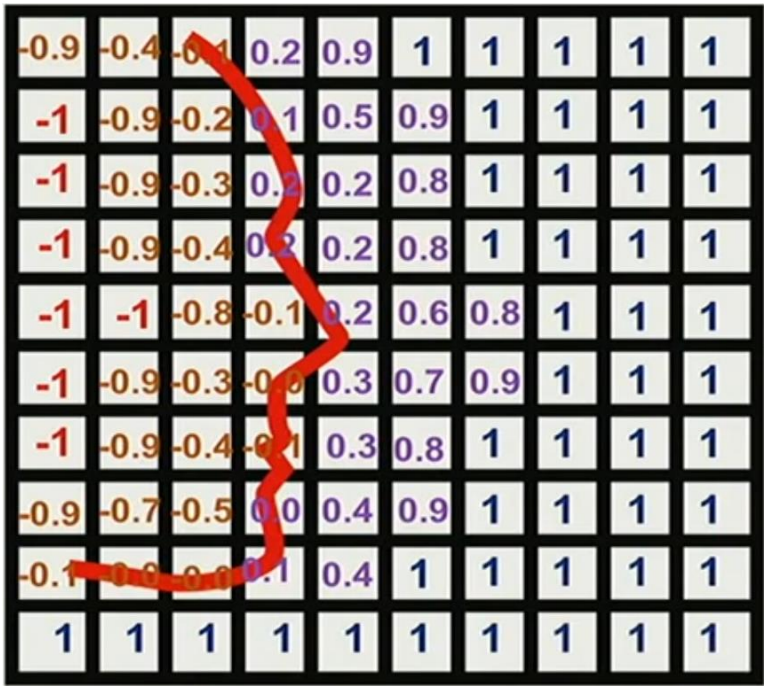

Figura 7: Esempio 2D di TSDF

E' facile notare che la superficie reale cadrà sui voxel in cui la TSDF cambia di segno (attraversamento dello zero).

Nel caso in cui fosse disponibile in input anche l'immagine a colori della scena attualmente ripresa, il contenuto dei pixel verrebbe anch'esso integrato all'interno del volume, aggiornandolo in modo analogo al valore di TSDF. Ciò non influirebbe comunque con i passaggi dell'algoritmo indicati in precedenza.

1.2.4 Raycasting

Infine, l'operazione di raycasting è utilizzata per fornire all'utente una visione della scena 3D ricostruita.

Ogni thread GPU, in parallelo, scorre lungo un raggio e renderizza un singolo pixel nell'immagine di output. Data una posizione iniziale e una direzione ogni thread GPU attraversa i voxel lungo il raggio estraendo le posizioni della superficie osservando i punti di attraversamento dello zero dei valori della TSDF salvati in essi. La superficie finale è calcolata utilizzando una interpolazione trilineare tra i punti trovati tramite ricerca dell'attraversamento dello zero. La normale alla superficie può essere direttamente calcolata sulla TSDF nei punti di attraversamento dello 0. Questi parametri sono utilizzati per il rendering della luminosità della superficie nell'immagine di output. Alternativamente, potranno essere utilizzati anche i valori RGB per visualizzare la scena a colori, nel caso l'algoritmo avesse abilitato questa opzione e ricevuto in input, assieme all'immagine depth, le immagini a colori corrispondenti.

Capitolo 2
CUDA (Compute Unified Device Architecture)

Come abbiamo visto nel capitolo precedente, KinectFusion è un algoritmo che necessita di avere caratteristiche real-time. Per ottenere ottimi risultati in termini di efficienza, infatti, ognuno dei passaggi descritti nel capitolo precedente gode di algoritmi ben studiati ed implementati in codice CUDA. Sia le operazioni svolte sulle mappe di vertici, sia quelle svolte sui singoli voxel sono tra loro indipendenti per ogni punto nel primo caso e per ogni voxel nel secondo, di conseguenza rappresentano operazioni fortemente parallelizzabili per cui CUDA e la programmazione GPGPU forniscono uno strumento utile per effettuare tali operazioni con la massima efficienza.

Il linguaggio di programmazione CUDA è una variante del C ed è stato fornito dalla NVIDIA al fine di sfruttare la potenza di calcolo delle proprie GPU (*Graphics Processing Unit*). Tale tipo di programmazione è chiamata GPGPU (*General-Purpose computing on Graphics Processing Units*) e si pone l'obiettivo di sfruttare la potenza computazionale dei processori grafici delle moderne schede video al fine di svolgere compiti di carattere generale piuttosto che essere utilizzati al solo fine di rendering di oggetti bidimensionali o tridimensionali. Faremo quindi una panoramica sugli aspetti (software e hardware) che riguardano lo sviluppo di codice in CUDA, con lo scopo di incrementare le prestazioni degli algoritmi sfruttando la parallelizzazione tipica delle attuali GPU.

In questo capitolo viene fornita una vista sull'architettura delle moderne GPU, con esempi più specifici riferiti all'hardware utilizzato per questo lavoro di tesi, ovvero il tablet Shield della NVIDIA ed il suo processore grafico Tegra K1.

2.1 Architettura delle GPU

Uno schema dell'architettura delle moderne GPU NVIDIA (che supportano CUDA) è riportato in **Figura 8**. Fondamentalmente il chip è composto da un insieme di multiprocessori, chiamati *Streaming MultiProcessor* (SM), i quali svolgono le operazioni computazionali. La quantità di SM in ogni chip dipende dalla fascia prestazionale di ogni GPU. Ogni SM è formato a sua volta da un numero variabile di *Stream Processors* (SP), commercialmente chiamati *core*. L'SP è l'unità fondamentale del chip ed è in grado di eseguire un'operazione matematica di base come: addizione, moltiplicazione, sottrazione ed altro, su interi o su numeri in virgola mobile.

Nello SM è compresa anche una memoria condivisa, accessibile da tutti gli SP, nonché delle cache per le istruzioni e per i dati e un'unità di decodifica delle istruzioni.

Sulla scheda sono inoltre presenti altri tipi di memoria, che però non sono privati per ogni SM ma sono accessibili da ognuno di essi e forniscono la locazione principale dove salvare grandi quantità di dati sulla GPU.

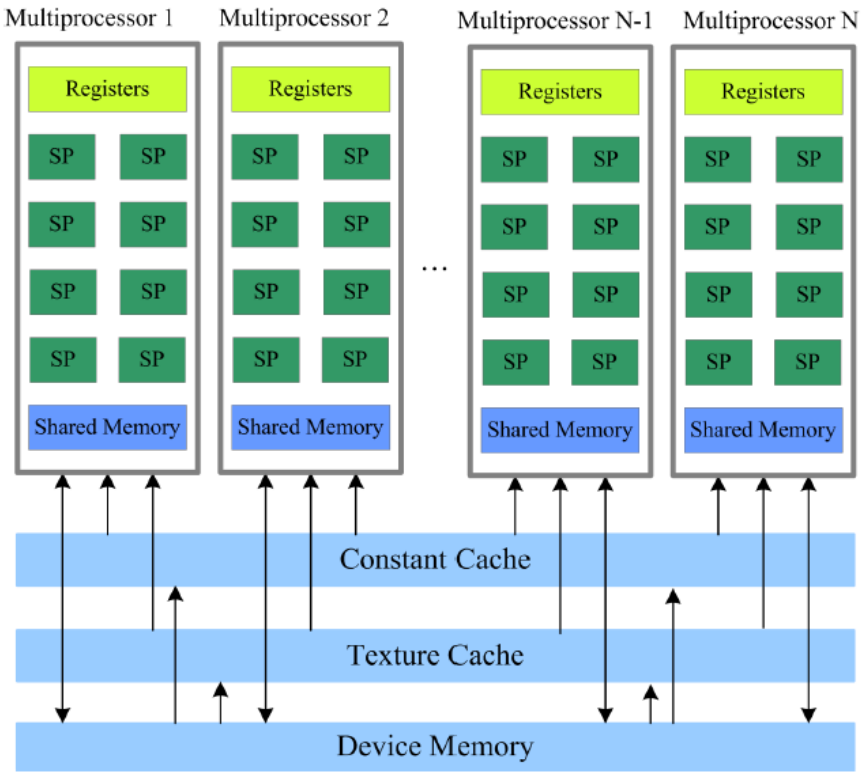

Figura 8: Architettura di una GPU

I processori delle moderne GPU lavorano a una frequenza di circa 1 Ghz, mentre le CPU hanno già da un po' di tempo raggiunto (e oltrepassato) i 3 Ghz. Questo non vuol dire però che le GPU siano più lente delle CPU. Infatti, queste ultime nelle moderne implementazioni raggiungono al più 8 core, mentre le GPU sono composte da centinaia di core. La differenza tra le tue tipologie di processori risiede appunto negli obiettivi che si prepongono. Infatti, le CPU tendono a minimizzare la latenza nell'eseguire in serie le operazioni, o al più con un parallelismo di 8 thread nel caso dell'octa-core, mentre le GPU tendono a massimizzare il throughput delle operazioni, ovvero il numero di thread eseguiti contemporaneamente [4, 5].

2.2 La programmazione con CUDA

La comprensione del modello di programmazione CUDA è la base per imparare ad utilizzare correttamente la GPU al fine di ottimizzare ed accelerare l'esecuzione dei nostri algoritmi. Se un qualsivoglia algoritmo esegue delle operazioni fortemente parallelizzabili è possibile aumentarne notevolmente l'efficienza riscrivendo tali operazioni in codice CUDA e sfruttando la GPU; tale incremento prestazionale ha il costo di una ovvia e necessaria ristrutturazione del codice nei pattern di programmazione CUDA. Vediamo di seguito i metodi messi a disposizione da questo pattern di programmazione.

2.2.1 I kernel

L'elemento principale della programmazione CUDA è il *kernel*. In CUDA, il *kernel* è quella funzione atta a contenere il codice parallelizzabile del nostro algoritmo. Una funzione diventa *kernel* se alla sua dichiarazione si prepone la parola chiave *__global__*.

Ad esempio: `__global__ void myMethod( ... ) { ... }`

Per eseguire le operazioni in parallelo, diverse istanze del *kernel* sono associate a specifici *thread*, la cui organizzazione è spiegata nel paragrafo successivo.

2.2.2 La griglia e i blocchi

In CUDA i thread vengono organizzati in modo ben strutturato. Questo sistema di organizzazione, per ogni *kernel*, fa uso di due elementi tridimensionali, il primo chiamato *grid* (o griglia) ed il secondo chiamato *block* (o blocco). All'interno di ogni cella della griglia è situato un blocco. Ad ogni posizione nelle tre dimensioni del blocco corrisponde un *thread*, come in **Figura 9**.

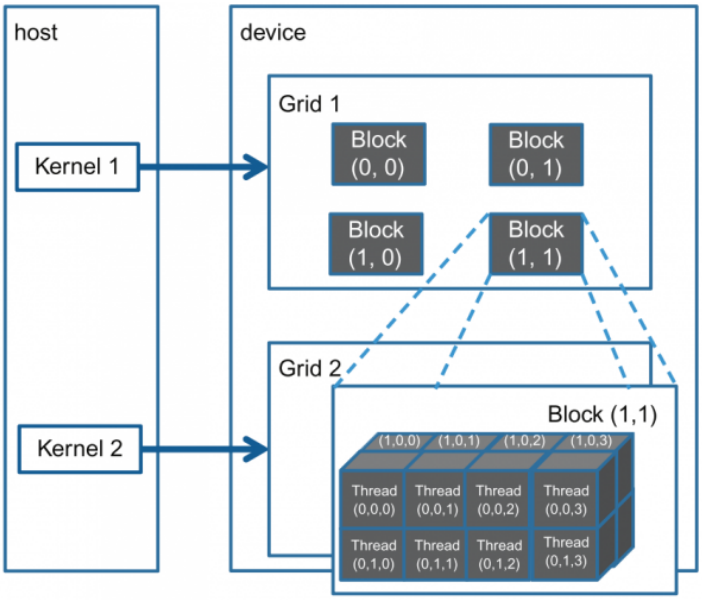

Figura 9: Composizione di griglia e blocchi per ogni kernel

Quindi la funzione *kernel*, per essere eseguita, al momento del lancio deve prendere in ingresso i due parametri riguardanti le grandezze di griglia e blocco così da riuscire a organizzare la disposizione dei *thread*, per esempio come nel seguente codice:

```
dim3 block_size(x, y, z);
dim3 grid_size(i, j, k);
myMethod <<< grid_size , block_size >>> (params);
```

2.2.3 Gestione efficiente dei thread

Per ottenere un'accelerazione GPU efficiente, le grandezze della griglia e dei blocchi vanno opportunamente settate in funzione della versione di compute

capability del proprio chip grafico. La compute capability è un valore che identifica una determinata versione di GPU e le sue specifiche caratteristiche.
La compute capability del processore grafico *K1* del tablet *Shield* utilizzato nella tesi è la versione 3.2, le cui principali caratteristiche possono essere osservate in **Figura 10**.

	Compute Capability							
Technical Specifications	**2.x**	**3.0**	**3.2**	**3.5**	**3.7**	**5.0**	**5.2**	**5.3**
Maximum number of resident grids per device (Concurrent Kernel Execution)	16		4	32				16
Maximum dimensionality of grid of thread blocks	3							
Maximum x-dimension of a grid of thread blocks	65535	2^{31}-1						
Maximum y- or z-dimension of a grid of thread blocks	65535							
Maximum dimensionality of thread block	3							
Maximum x- or y-dimension of a block	1024							
Maximum z-dimension of a block	64							
Maximum number of threads per block	1024							
Warp size	32							
Maximum number of resident blocks per multiprocessor	8	16				32		
Maximum number of resident warps per multiprocessor	48	64						
Maximum number of resident threads per multiprocessor	1536	2048						
Number of 32-bit registers per multiprocessor	32 K	64 K			128 K	64 K		
Maximum number of 32-bit registers per thread block	32 K	64 K						32 K
Maximum number of 32-bit registers per thread	63	255						
Maximum amount of shared memory per multiprocessor	48 KB				112 KB	64 KB	96 KB	64 KB
Maximum amount of shared memory per thread block	48 KB							

Figura 10: Caratteristiche delle varie versioni di compute capability

In questo tipo di architettura, uno o più blocchi assegnati ad un SM vengono partizionati in *warp* e successivamente schedulati da un *warp scheduler*. Indipendentemente dalla versione di compute capability, come è possibile vedere dalla **Figura 10** la dimensione del *warp* è sempre di 32 *thread*, eventuali warp parziali sono riempiti con *thread* fittizi. Per comprendere meglio quali considerazioni vanno fatte per strutturare i *thread* in maniera efficiente, prendiamo in considerazione i limiti hardware imposti dalla compute capability (versione 3.2) del nostro processore grafico *K1*.
Dalla tabella in **Figura 10** è possibile notare che:

- Il numero massimo di thread per SM è 2048;
- Il numero massimo di thread per blocco è 1024;
- Il numero massimo di blocchi per SM è 16.

Data la divisione fissa in *warp* da 32 *thread* è buona norma che il blocco abbia un numero di *thread* multiplo di 32, in maniera tale da evitare la formazione di *warp* parzialmente pieni. Con i limiti di capacità sopra riportati, ogni SM può

contenere al massimo 2048 / 32 = 64 warp, questo implica il fatto che bisogna assegnare un numero di *thread* per blocco tale che il numero di *warp* effettuati su tali *thread*, sia divisore di 64, così da permettere all'SM di essere completamente riempita dai *warp*. Il numero totale non può però essere 64 stesso poiché un tale numero di *warp* contenenti 32 thread ciascuno risulterebbe in 2048 *thread* per blocco, mentre il valore massimo di *thread* per blocco nella nostra specifica compute capability è di 1024.

Inoltre, dalla considerazione del vincolo sul massimo di blocchi per SM, e quello sul massimo dei *warp* per SM, possiamo notare che ogni blocco non dovrebbe contenere meno di 64 warpMax / 16 blocchiMax = 4 warp per ogni blocco, ovvero, 4 * 32 = 128 *thread* per blocco.

Date le considerazioni fatte sopra, quindi, i numeri utili di *warp* per una corretta distribuzione dei *thread* sono tutti i valori maggiori o uguali a 4 e divisori di 64 (escluso 64 stesso), perciò 4-8-16-32 *warp*, che corrispondono rispettivamente a 128-256-512-1024 *thread* per blocco.

2.3 La memoria

Come è possibile vedere nella **Figura 8** all'inizio di questo capitolo, il sistema CUDA è formato anche da diversi tipi di memorie. Le più importanti e più utilizzate sono la Global Memory e la Shared Memory. Dono presenti anche altri tipi di memorie come quelle in sola lettura, Constant Memory e Texture Memory, il cui utilizzo è riservato a casi particolari in cui l'applicazione possa avere la necessità di meccanismi veloci di reperimento di determinati dati e meccanismi di caching. Infine, sono presenti i registri, ovvero memorie dedicate al singolo thread e con visibilità locale, in modo simile a quanto presente nelle CPU.

2.3.1 La Global Memory

La Global Memory è la memoria di più grande dimensione sulla GPU e può variare da poche centinaia di MB a diversi GB (nelle schede più performanti). Essa rappresenta il punto principale dove inserire tutti i dati che servono al

corretto svolgimento dell'applicazione. Questa memoria, come già detto, ha il pregio di avere una grande dimensione ma il difetto conseguente di presentare tempi di accesso in lettura e scrittura considerevolmente più elevati rispetto a memorie di dimensione minore. Per tale motivo, quando possibile è preferibile spostare i dati nelle altre memorie per incrementare la velocità di esecuzione. Come è visibile in **Figura 8** questa è l'unica memoria ad avere una visibilità globale ed è quindi accessibile da tutti i thread della griglia, sia in lettura che in scrittura.

Per utilizzare la Global Memory si deve allocare lo spazio necessario ad ospitare i dati ed eventualmente copiare gli stessi da CPU a GPU o utilizzare direttamente codice CUDA che abbia lo scopo di operare su tali dati. Il linguaggio CUDA fornisce le primitive necessarie per effettuare queste operazioni. Per esempio, lo spazio in GPU potrebbe essere allocato in questo modo:

```
float n = 5;
size_t  size = n * sizeof(float);
float* device_A;
cudaMalloc(&device_A , size);
```

Questo codice alloca sulla GPU uno spazio pronto per ospitare 5 valori *float*. Successivamente copiamo quindi i valori dalla CPU alla GPU, in questo modo:

```
float* host_A = (float*) malloc (size);
cudaMemcpy(device_A , host_A , size, cudaMemcpyHostToDevice);
```

Al termine della computazione su GPU copiamo il risultato sulla CPU:

```
cudaMemcpy(host_A , device_A , size, cudaMemcpyDeviceToHost);
```

2.3.2 La Shared Memory

Un'altra memoria molto utile nella programmazione CUDA è la Shared Memory, cioè la memoria condivisa tra i thread che stanno all'interno della stessa SM. Questa memoria è molto piccola in confronto alla Global Memory, infatti, per esempio, nella compute capability del processore grafico *Tegra K1* è di soli 48KB, ma permette un accesso molto più veloce ai thread a cui è visibile. Per creare una variabile in Shared Memory è sufficiente preporre alla sua dichiarazione la parola chiave __shared__. Il ciclo di vita di una variabile shared

è limitato al ciclo di vita del kernel in cui è stata creata. Una variabile *shared* è utile, quindi, per scambiare informazioni tra i thread all'interno di un blocco. Infatti, all'interno dello stesso SM si hanno diverse copie di questa variabile, una per ogni blocco.

2.4 Un esempio di codice CUDA

Uno degli esempi più veloci da mostrare per comprendere la logica del modello CUDA, è quello della somma tra elementi di indici corrispondenti tra due vettori. In una normale programmazione su CPU, avremmo un ciclo iterativo all'interno del quale ad ogni iterazione viene fatta la somma tra tue elementi dei vettori: *c[i] = a[i] + b[i]*. In programmazione CUDA invece, sarà istanziato il *kernel* le cui operazioni interne vengono svolte, in diverse istanze, da più thread contemporaneamente.
Lo vediamo in questo esempio di codice:

```
__global__ void sommaArray(float *A, float *B, float *C)
{
      int idx;
      idx = blockIdx.x*blockDim.x + threadIdx.x;
      *(C+idx) = *(A+idx) * *(B+idx);
}
```

Come è possibile notare da questo esempio di codice, ogni thread deve recuperare il proprio ID. Tale ID può essere ottenuto tramite le primitive messe a disposizione dal linguaggio CUDA, nel caso specifico (blockIdx.x, blockDim.x e threadIdx.x). Nell'esempio che abbiamo mostrato, è stato supposto che i thread siano situati nel blocco solo lungo la direzione x, ma come è stato detto in precedenza, il blocco ha una struttura tridimensionale, quindi esistono per tali variabili anche le versioni per le direzioni y e z. Tali primitive, sono utili al programmatore per identificare un thread nello scenario CUDA. Vediamo un piccolo dettaglio su queste primitive:

- blockIdx.x indica l'ID del blocco nella direzione x della griglia.
- blockDim.x indica di quanti thread è composto il blocco nella direzione x.
- threadIdx.x indica l'ID del thread all'interno del blocco.

Quindi, per indicizzare in maniera efficiente la somma all'interno del vettore, sfruttiamo l'ID globale dello specifico thread, questo si ottiene, sommando l'ID che il thread ha all'interno del blocco al prodotto tra l'ID del blocco e la sua dimensione, in maniera tale da utilizzare questo prodotto come offset per l'ID globale del thread.

In questo modo, ogni thread può lavorare separatamente e contemporaneamente su un elemento del vettore diminuendo drasticamente il tempo di computazione totale.

2.5 Il tablet NVIDIA Shield

Il dispositivo mobile utilizzato per questa tesi è il tablet Shield (visibile nella **Figura 11**) della NVIDIA. La scelta è ricaduta su questo dispositivo, data la natura del tipo di codice utilizzato da KinectFusion, ovvero CUDA. Infatti, per poter utilizzare un software che sfrutti l'accelerazione grafica offerta da CUDA è necessario che l'architettura del chip grafico utilizzato sia compatibile con tale linguaggio di programmazione. Nello scenario dei dispositivi mobili che sono oggi in commercio, il tablet Shield della NVIDIA è uno dei più recenti ad offrire un tale tipo di GPU. La GPU in questione è la *Tegra K1*, la quale offre 192 cuda cores a 950 Mhz ed una compute capability alla versione 3.2 (**Figura 10**).

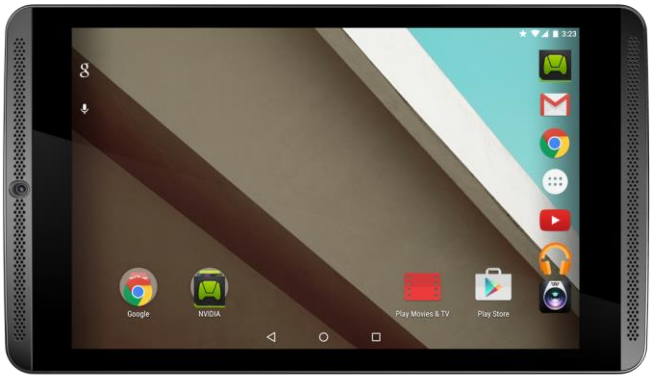

Figura 11: Tablet NVIDIA Shield

Capitolo 3
Dipendenze dell'applicazione Android

In questo capitolo verranno descritte le librerie di supporto che sono state utilizzate per effettuare il porting dell'algoritmo KinectFusion in ambiente mobile Android. Nel descrivere tali librerie, verrà anche illustrato uno dei software di supporto alla programmazione, il quale è stato di fondamentale importanza al fine di eseguire una corretta compilazione del codice di KinectFusion in ambiente multipiattaforma.

Inizieremo questo capitolo descrivendo come vengono acquisite le immagini di depth all'interno dell'applicazione, tramite l'utilizzo di una libreria esterna chiamata *InputOutputLibray*.

Analizzeremo in seguito la procedura ed i software utilizzati per poter compilare l'algoritmo KinectFusion su ambiente Android. Ciò porterà alla creazione di alcune librerie che costituiranno il core dell'applicazione.

Successivamente, vedremo come è stato possibile interfacciare il codice C di KinectFusion all'interno dell'applicazione Android, la quale è stata scritta in codice Java. Verrà perciò descritto il framework JNI tramite il quale è possibile effettuare questo interfacciamento, che in tal caso farà riferimento alle librerie KinectFusion precedentemente compilate.

Infine, verrà fatta una panoramica sulla libreria *OpenCV*, la quale contiene gli strumenti fondamentali per l'elaborazione delle immagini e che viene utilizzata da tutti i diversi componenti del progetto.

3.1 InputOutputLibrary

L'algoritmo KinectFusion, per poter eseguire, necessita di ricevere in input una immagine depth ed eventualmente una immagine a colori. Per questo motivo, nell'analizzare l'insieme delle librerie che costituiscono il supporto all'applicazione Android sviluppata per questa tesi, è sembrato opportuno cominciare descrivendo la libreria che permette l'acquisizione di tale immagine.

La *InputOutputLibrary* è la libreria che è stata utilizzata per caricare all'interno del nostro software immagini depth ed RGB estratte da un dataset oppure acquisite da un apposito sensore. Nel nostro caso è stato utilizzato il sensore Structure, appositamente realizzato per dispositivi mobili ed il quale permette di ottenere depth map in una risoluzione massima di 640x480. Per le immagini a colori la libreria effettuerà un mapping software sfruttando una previa calibrazione del sensore.

Per poter iniziare l'acquisizione delle immagini all'interno del software, bisogna inizializzare un oggetto Grabber, istanza di una delle due classi che rappresentano le tipologie di input messe a disposizione da questa libreria. Le classi in questione sono *ImageGrabber* e *StructureGrabber*. *ImageGrabber* è la classe che si occupa di acquisire immagini da dataset. Nel nostro caso questa classe supporta il famoso dataset TUM [3] per quanto concerne il formato delle immagini ed è in grado di leggerle dopo aver ricevuto in input un file in cui sono indicati i path alle coppie di immagini depth ed RGB. *StructureGrabber* è invece la classe utilizzata per acquisire le immagini dal sensore Structure ed effettuare in modo trasparente il mapping fra immagine depth ed immagine RGB. L'elemento fondamentale è che entrambe le classi implementano l'interfaccia *SensorGrabberBGRD*, che definisce le funzionalità principali che entrambe le classi devono necessariamente implementare per gestire in maniera adeguata l'acquisizione delle varie immagini. La **Figura 12** mostra il diagramma delle classi con gli elementi fondamentali presenti nella libreria.

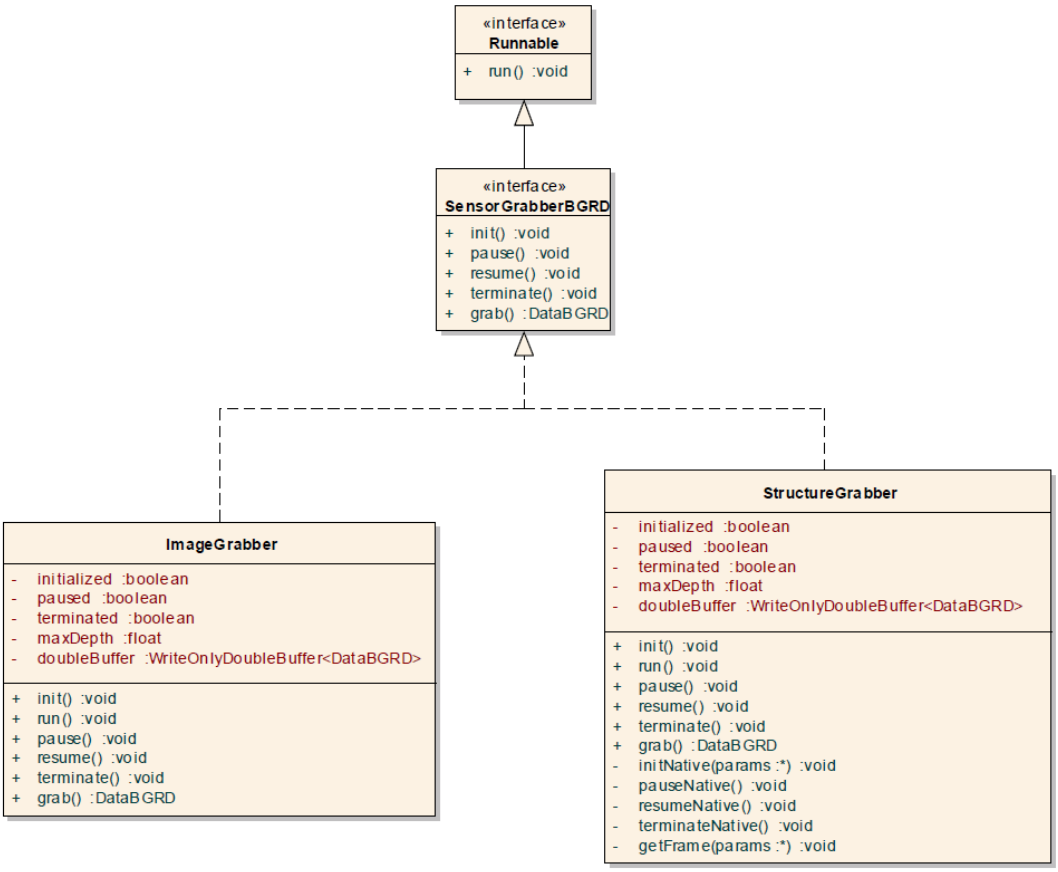

Figura 12: Diagramma delle classi Grabber

Dalla figura si può notare che nell'interfaccia sono specificati i seguenti metodi:

- *init()*: Inizializza l'oggetto grabber dipendentemente dall'implementazione. Nel caso del dataset questa inizializzazione non farà altro che leggere un descrittore che indica i path alle immagini da acquisire, mentre nel caso del sensore Structure si effettuerà l'inizializzazione del dispositivo con alcuni parametri di configurazione, anch'essi prelevati da file di testo;
- *pause()*: Mette in pausa l'acquisizione delle immagini;
- *resume()*: Fa ripartire l'acquisizione;
- *terminate()*: Termina il grabbing dei frame e rende perciò l'oggetto non più valido;
- *grab()*: Utilizzato per estrarre effettivamente l'immagine depth in base all'implementazione dell'interfaccia. Se l'immagine è estratta da dataset non fa altro che leggerla da una directory apposita e salvarla in memoria,

mentre se l'immagine è acquisita da sensore verrà prelevata tramite codice nativo C. Il metodo restituisce un oggetto di tipo *DataBGRD*. Tale oggetto è definito all'interno della libreria e, oltre a contenere informazioni come altezza e larghezza delle immagini, timestamp, e massima misura di profondità, può contenere sia la versione a colori (in questo caso in formato BGR) sia la versione depth (D) dell'immagine, salvate rispettivamente come un array di byte la prima (3 bytes per colore) e come array di float la seconda.

L'acquisizione dei frame può avvenire in due modi diversi. Il primo concerne l'esecuzione del metodo *grab* sopra indicato da parte di un altro componente dell'applicazione, ottenendo così l'immagine depth e l'eventuale immagine RGB. Il secondo metodo riguarda invece l'esecuzione del grabber specifico all'interno di un thread indipendente. In questo caso i dati verranno prelevati da un buffer su cui il grabber andrà a scrivere.

Anche in questa libreria è presente del codice nativo, utilizzato principalmente nell'esecuzione dello *StructureGrabber*, come visibile dal diagramma delle classi (i metodi che terminano con la parola *Native* sono in realtà richiamati dai metodi pubblici Java e fanno riferimento a codice C). Per poter quindi eseguire il codice contenuto nella *InputOutputLibrary* è necessario caricare alcune librerie C/C++. In questo caso ciò avviene invocando il metodo statico *initLibrariesAsync* della classe *SensorLib*, specificando un oggetto callback della classe *LibrariesLoadedCallback* su cui verrà invocato il metodo *onLibrariesLoaded* una volta che le librerie sono state caricate correttamente. In questo modo l'applicazione principale potrà porsi in attesa dell'esecuzione della callback prima di effettuare l'inizializzazione del grabber, in quanto altrimenti l'esecuzione dei successivi metodi provocherebbe un errore.

3.2 CMake (Cross Platform Make)

CMake è un software multipiattaforma che consente la gestione di grandi progetti favorendo la creazione di file di configurazione per facilitare la compilazione su diverse piattaforme.

Tale software di supporto alla programmazione fa uso di un proprio file di configurazione chiamato «CMakeLists.txt» nel quale, con un suo semplice linguaggio di programmazione, è possibile indicare tutti i moduli che compongono un progetto, le loro dipendenze e diversi parametri di compilazione. Appoggiandosi a tale file di configurazione CMake è in grado di creare per ogni piattaforma e in maniera automatica il Makefile su Linux o file di configurazione per importare correttamente i progetti su vari ambienti di sviluppo, come ad esempio l'IDE Eclipse utilizzato per questa tesi.

3.2.1 CMake-GUI

Per poter creare in maniera automatica i file di configurazione del progetto, Cmake naviga all'interno del file «CMakeLists.txt» trovando tutte le dipendenze e parametri di compilazione di cui il progetto necessita. Con il crescere della complessità del progetto, delle dipendenze e dei parametri di compilazione, inserire da riga di comando tutti i riferimenti alle componenti necessarie potrebbe diventare un'operazione piuttosto lunga. Inoltre, ciò porterebbe alla creazione di lunghe istruzioni di difficile lettura e revisione nel caso fossero presenti errori.
Per questo motivo CMake mette a disposizione del programmatore un'interfaccia grafica che elenca e schematizza tutti gli elementi trovati nel «CMakeLists.txt» necessari alla buona riuscita dell'operazione di creazione dei file di configurazione. In questo modo è possibile procedere a specificare le varie componenti in maniera più rapida e a correggere velocemente eventuali errori di impostazione.

3.2.2 Importazione su Eclipse di KinectFusion con CMake-GUI

Al fine poter di effettuare con maggiore facilità eventuali modifiche al codice di KinectFusion e per una sua migliore gestione, è stato deciso di importare il progetto all'interno dell'ambiente di programmazione Eclipse. Per fare ciò è stato usato il software di supporto alla programmazione CMake, tramite la sua interfaccia grafica.

Una volta entrati da terminale nella cartella del progetto KinectFusion contenente il file «CMakeLists.txt», si può aprire l'interfaccia grafica di CMake semplicemente lanciando il comando *cmake-gui*. Essendo CMake un software di supporto al cross-compilazione, l'interfaccia grafica chiederà in prima istanza per quale ambiente di programmazione generare i file di configurazione, tramite la finestra in **Figura 13**.

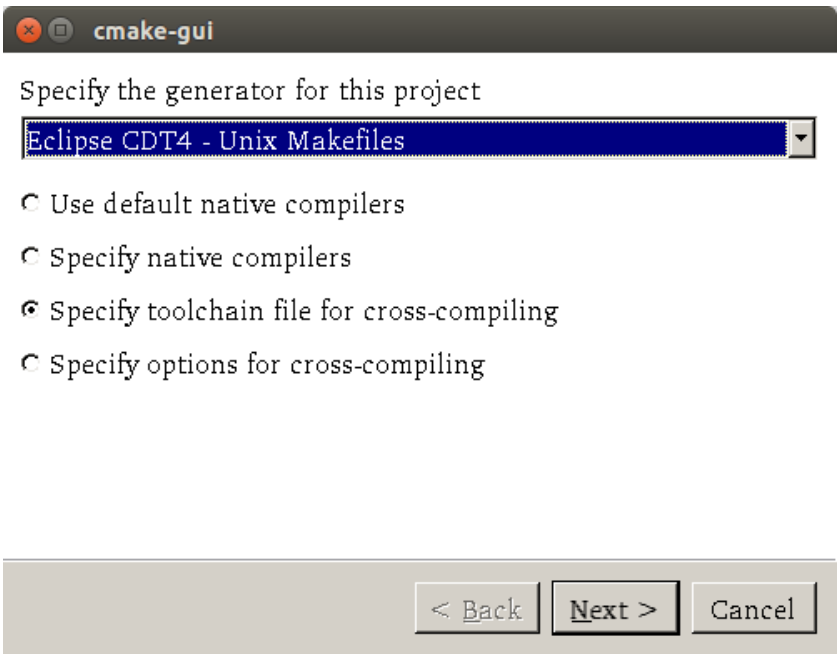

Figura 13: Scelta del tipo di progetto da generare

Nel nostro caso scegliamo Eclipse con Makefile per sistemi Unix, e indichiamo inoltre di voler specificare il toolchain file per la cross-compilazione che intendiamo effettuare (arm-linux-android). Questo file di toolchain è un particolare file di testo che permette di settare automaticamente dei parametri del CMake a seconda, appunto, del tipo di toolchain utilizzato. In questo caso il file si chiama «android.toolchain.cmake» ed è presente già in diversi progetti compilabili con Android, come ad esempio OpenCV.

Eseguito questo primo passaggio, l'interfaccia che si presenta è quella in **Figura 14**.

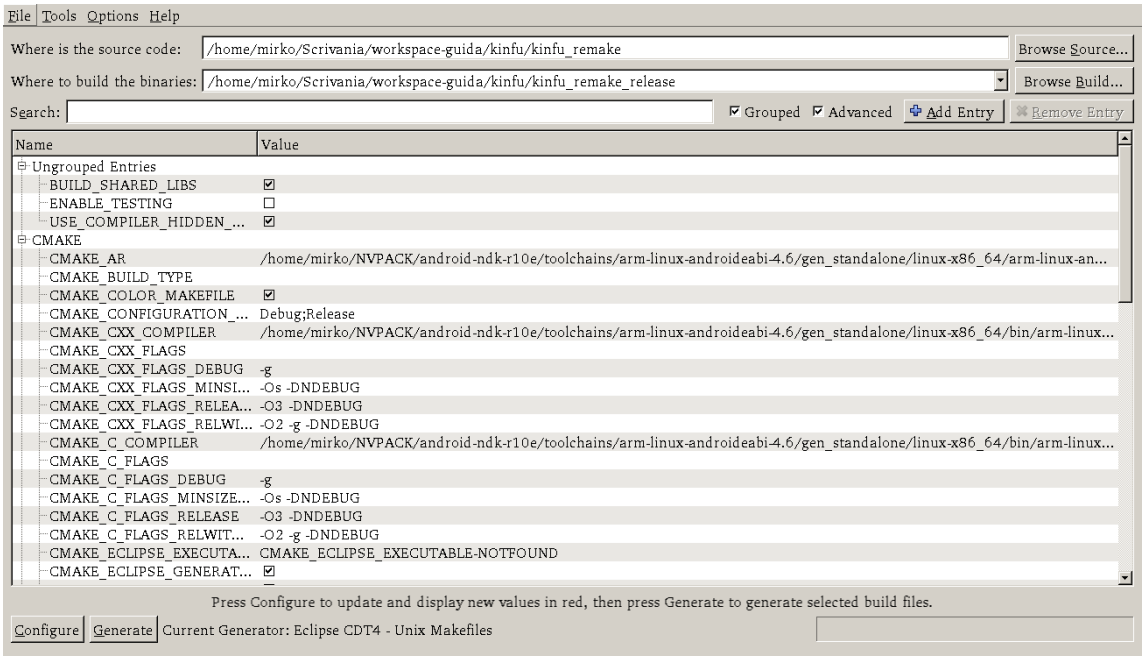

Figura 14: Interfaccia di CMake

Come è possibile vedere nella figura, l'interfaccia è costituita nella parte alta da due campi che riguardano il primo la cartella del codice sorgente ed il secondo la cartella di build. Nel nostro progetto sono state create entrambe nella stessa directory padre per evitare confusione. Nel corpo principale dell'interfaccia, invece, sono indicate tutte le dipendenze trovate da CMake nel «CMakeLists.txt» elencate per categorie. Per il nostro progetto, le categorie principali della configurazione sono *CMAKE* (visibile in figura), *CUDA*, *OpenCV* (parzialmente utilizzato nell'attuale implementazione dell'algoritmo) e *ANDROID* (quest'ultima presente grazie al toolchain file inserito in precedenza).

La **Figura 14** mostra in realtà l'immagine dell'interfaccia di CMake, dopo aver già effettuato l'impostazione dei parametri, in maniera congrua al proprio ambiente di lavoro, per quanto concerne i path ai vari elementi, quali cartelle sorgenti e di build, i compilatori da utilizzare e le diverse variabili necessarie ai fini della compilazione. Nella categoria *CMAKE* di particolare importanza è la variabile *CMAKE_INSTALL_PREFIX*, nella quale sarà settato il path alla cartella che abbiamo chiamato *kinfu_remake_installed*, situata nella stessa directory della cartella con i file sorgenti e la cartella di build, e nella quale dopo il processo di

compilazione verranno create le shared libraries da importare nell'applicazione Android.

Come abbiamo appena detto, l'operazione di configurazione dei path dei moduli necessari alla compilazione comprende anche le categorie *CUDA* e *OpenCV*, come è mostrato nelle due immagini seguenti (**Figura 15** e **16**).

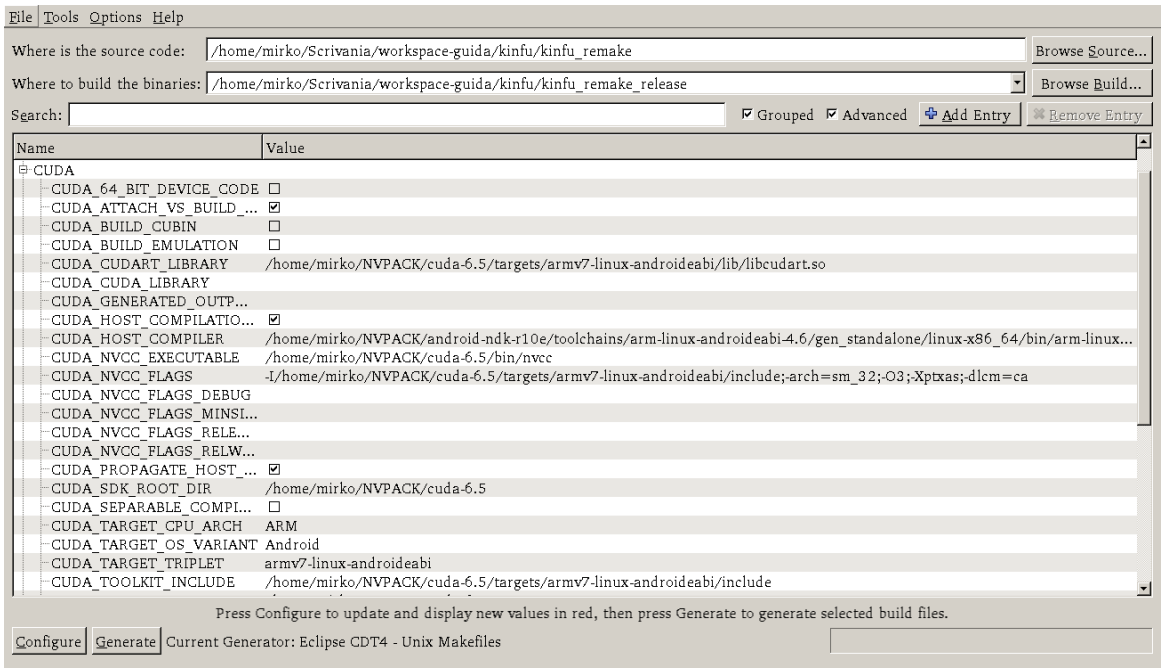

Figura 15: Setting CMake per CUDA

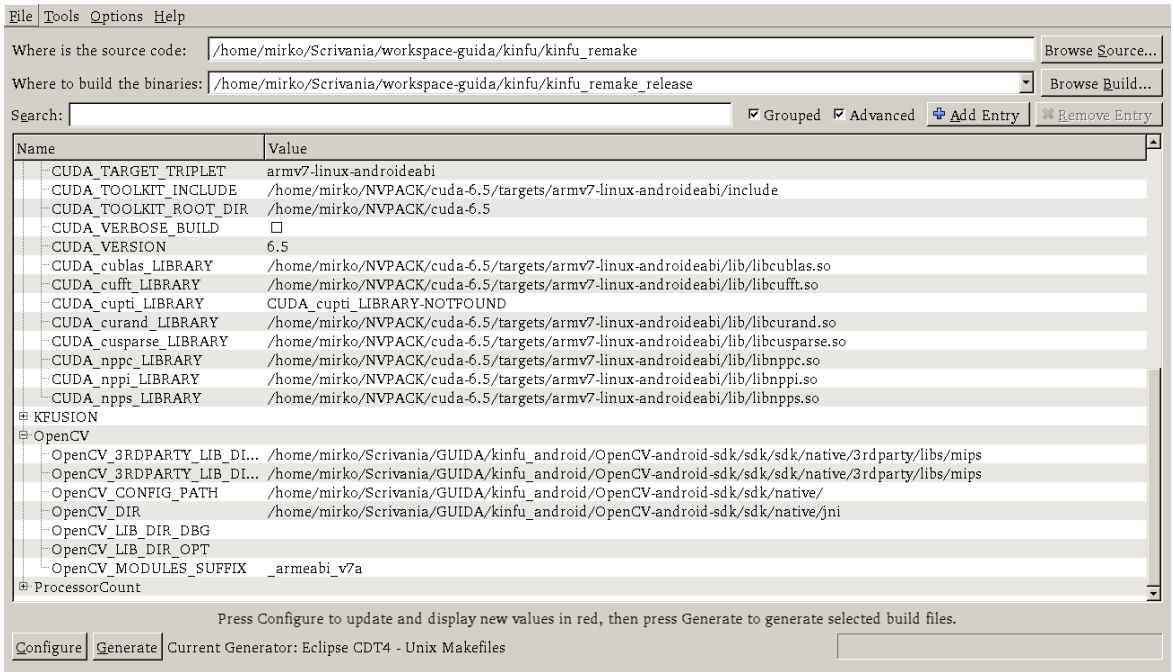

Figura 16: Setting CMake per OpenCV

Per quanto riguarda *CUDA*, è stato necessario settare le seguenti cose (**Figura 15**):

- Il path al compilatore NVCC;
- I parametri di cross-compilazione per Android in architettura ARMv7 (architettura CPU utilizzata dal tablet Shield di riferimento), ovvero le variabili *CUDA_TARGET_CPU_ARCH*, *CUDA_TARGET_OS_VARIANT* e *CUDA_TARGET_TRIPLET*;
- I flag del compilatore NVCC, tra cui il riferimento agli include del toolkit CUDA ed alcuni parametri di ottimizzazione;
- I path ad una serie di librerie CUDA specifiche per l'architettura del processore ARMv7, l'unica attualmente supportata nel toolkit di sviluppo CUDA per Android. Diverse librerie sono in realtà opzionali, in quanto l'unica strettamente richiesta è la libreria «libcudart.so».

Per quanto riguarda *OpenCV* (**Figura 16**), invece, sono stati settati i path specifici alla libreria, ed è stato inserito il suffisso per l'architettura del processore ARMv7.

L'ultima categoria tra quelle presenti è quella *ANDROID* che deriva dal file di toolchain inserito e nella quale sono specificati i parametri fondamentali alla cross-compilazione. Tra le variabili di questa categoria è stato necessario aggiungere manualmente la variabile *ANDROID_TOOLCHAIN_NAME* per specificare il giusto tipo e versione di compilatore per la cross-compilazione desiderata, visibile in **Figura 17**. Le altre impostazioni erano già presenti in automatico con possibilità di modifica in base alle preferenze.

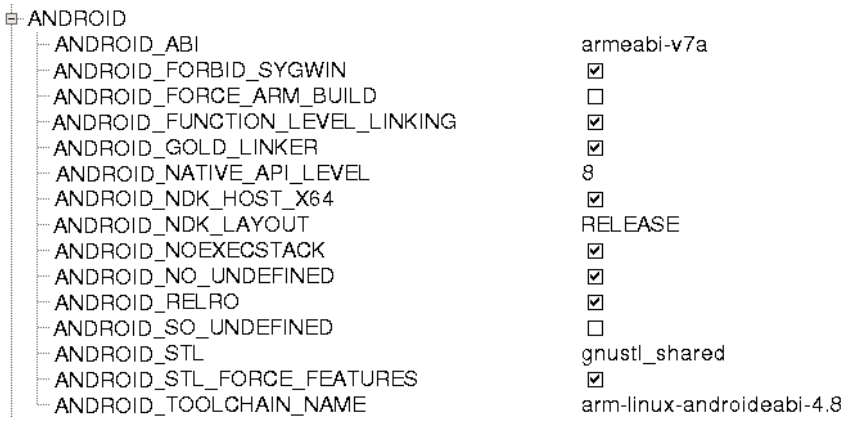

Figura 17: Setting ANDROID per la cross-compilazione

Cliccando sui pulsanti dell'interfaccia di CMake «Configure» e successivamente «Generate», viene automaticamente creato il progetto all'interno della cartella di build precedentemente impostata, insieme ai file di configurazione compatibili con Eclipse.

Adesso è possibile importare il progetto KinectFusion dalla cartella di build in Eclipse con la procedura di import standard e successivamente lanciare il *build* (che esegue make) del progetto ottenendo i file .so delle shared libraries nella cartella lib (sotto-cartella di quella di build). Successivamente è possibile eseguire, nella cartella di build, il comando *make install*, il quale crea all'interno della cartella *kinfu_remake_installed* (il cui path è stato impostato precedentemente nella variabile *CMAKE_INSTALL_PREFIX* della gui di CMake) la cartella *include* contenente tutti gli header utili per l'interfacciamento con l'algoritmo e la cartella *lib* contenente una copia delle librerie shared. Questi due path saranno successivamente utilizzati per includere le librerie nell'applicazione Android, unite agli header a cui fanno riferimento.

3.3 JNI (Java Native Interface)

Come è stato discusso nel Capitolo 2, ogni passo dell'algoritmo KinectFusion per essere eseguito efficientemente su GPU si avvale necessariamente di implementazioni in linguaggio CUDA, il quale è una variante del C. Di conseguenza, si ha la necessità di mantenere tali implementazioni CUDA anche nella versione mobile dell'algoritmo. Inoltre, il codice non CUDA che

l'algoritmo esegue è implementato in C/C++. D'altra parte però, l'applicazione Android sviluppata è basata sul linguaggio Java. Per consentire al codice Java dell'applicazione di sfruttare le funzionalità del codice C di KinectFusion è necessario ricorrere ad un framework apposito.

JNI (*Java Native Interface*) è un framework del linguaggio Java che si propone appunto di assolvere al compito di fare da ponte tra programmi Java e funzionalità scritte in linguaggio nativo al sistema in cui ci si trova ad operare, nel nostro caso specifico il linguaggio C o C++. Questo framework è disponibile su piattaforma Android e principalmente viene utilizzato in applicazioni che richiedono performance elevate oppure per gestire la compatibilità con algoritmi scritti in linguaggio C o C++.

3.3.1 Utilizzo di JNI

Vediamo adesso, come è possibile richiamare un metodo nativo dal codice Java di un'applicazione Android. Per effettuare una chiamata nativa, in primo luogo, bisogna creare una classe Java che contiene la dichiarazione del metodo nativo.

Il codice seguente crea una classe chiamata *TestConversion* che dichiara il metodo nativo *inToString*, identificato dalla keyword *native*:

```
public class TestConversion
{
      public native String intToString(int number);
}
```

Successivamente da linea di comando utilizzando l'utility *javah* è possibile, a partire dalla classe definita in precedenza, ottenere direttamente l'header per il codice C, che in questo caso riporta la seguente signature:

```
JNIEXPORT jstring JNICALL
Java_packageName_TestConversion_intToString
(JNIEnv *env, jobject obj, jint number)
```

Nella quale, jstring rappresenta il valore di ritorno della funzione nativa ed il *packageName* è il nome del package nel progetto all'interno del quale è definita la classe Java vista sopra. Da questo esempio si può notare come viene indicato il nome della classe ed il nome del metodo all'interno della signature della

funzione. Come parametri di input alla funzione invece possiamo vedere, oltre al jint utile ai nostri scopi, anche JNIEnv che rappresenta una struttura dati che permette di invocare le funzioni di utilità di JNI, come ad esempio creare array Java e gestire l'invocazione di metodi Java da C, e infine un parametro jobject che rappresenta l'oggetto su cui è invocato tale metodo (in questo caso un oggetto della classe TestConversion).

Dal punto di vista dell'organizzazione dei file, le implementazioni delle funzioni native vengono mantenute separate da quelle Android dell'applicazione Java. A questo proposito all'interno del progetto è predisposta una cartella chiamata *jni*, la quale contiene tutti i file header (visti sopra) e i file sorgenti C/C++ delle implementazioni. All'interno di tale cartella, è di fondamentale importanza il file «Android.mk», il quale è utilizzato per indicare il nome del modulo (il quale costituisce di fatto il nome della libreria C/C++ che verrà generata), i file sorgente in cui viene implementato ed eventualmente header ed ulteriori librerie da collegare. Nel nostro esempio, supponendo di aver implementato il metodo all'interno del file «intToString.cpp», per creare correttamente la libreria il file «Android.mk» dovrà contenere le seguenti linee di codice:

```
include $(CLEAR_VARS)

LOCAL_MODULE := int2string
LOCAL_SRC_FILES := intToString.cpp

include $(BUILD_SHARED_LIBRARY)
```

Infine, utilizzando il comando *ndk-build* viene effettivamente creata la libreria «libint2string.so» (formato del nome: lib + nome del modulo), utilizzabile nell'applicazione Android.

Per poter utilizzare la libreria appena generata all'interno dell'applicazione che si intende sviluppare, prima di creare un oggetto della classe TestConversion è necessario caricare la libreria attraverso il seguente codice:

```
System.loadLibrary("int2string");
```

3.3.2 Tipi di dato in JNI

Come è possibile vedere nel paragrafo precedente, nella dichiarazione del prototipo della funzione, il tipo di oggetto indicato come parametro di ritorno è jstring. JNI infatti utilizza nella sua implementazione del codice nativo dei particolari tipi di dato che corrispondono a quelli Java. Si hanno in questo modo le corrispondenze tra float e jfloat, int e jint, String e jstring e così via.

E' necessario però sottolineare in questo ambito la differenza tra i tipi nativi come *float* e *int*, accessibili direttamente nel codice nativo, e gli oggetti String o Array (per cui si ha l'apposito tipo jArray) che invece necessitano di alcuni metodi di JNI per essere manipolati.

Infatti, sarebbe un errore trattare, ad esempio, jstring come una normale stringa di caratteri (char*). Per questo motivo, JNI mette a disposizione dei metodi appositi, come ad esempio *GetStringUTFChars()*, per convertire la jstring in una stringa C per poi effettuarci sopra le operazioni desiderate. Questi metodi sono appunto invocabili tramite l'utilizzo del puntatore a JNIEnv.

3.4 La libreria OpenCV

OpenCV (*Open Source Computer Vision*) [2] è una libreria open-source multipiattaforma molto famosa nell'ambito dell'elaborazioni delle immagini. Questa libreria fornisce una serie di strutture dati e funzionalità che permettono di elaborare le immagini in modo efficiente. Per questi motivi, questa libreria è da supporto in tutte le fasi del progetto, inclusi l'algoritmo KinectFusion e la InputOutputLibrary. Nello specifico, all'interno di KinectFusion, la libreria viene utilizzata nella sua versione in codice C, mentre nella InputOutputLibrary essa viene utilizzata combinando codice C e Java.

Nell'applicazione Android sviluppata per questa tesi la libreria OpenCV è stata utilizzata per permettere la corretta elaborazione e visualizzazione sull'interfaccia utente dell'immagine ottenuta in seguito all'operazione di raycasting effettuata da KinectFusion.

In particolare, quando invochiamo KinectFusion all'interno della nostra applicazione, tramite l'ausilio di JNI, viene restituito un valore *long*

rappresentante il puntatore ad un oggetto *Mat*, in cui è salvata l'immagine della ricostruzione 3D. Infatti, per gestire le immagini, OpenCV mette a disposizione il tipo di dato *Mat* che può essere pensato come una matrice di valori, il cui numero di righe e colonne sono le dimensioni di altezza e larghezza dell'immagine. La classe *Mat* di OpenCV, nella sua versione Java, prevede anche un costruttore con il quale è possibile creare un oggetto di tale classe passando il puntatore di un altro oggetto *Mat* già inizializzato nel codice nativo C. In questo modo è possibile acquisire l'immagine della ricostruzione da codice C, essendo l'algoritmo KinectFusion implementato in tale linguaggio, ed utilizzarla poi all'interno della nostra applicazione nel codice Java. Successivamente, un'altra funzione di utility di OpenCV (*matToBitmap*) viene sfruttata per creare un oggetto Bitmap a partire dalla Mat. Tale oggetto Bitmap, tipico della piattaforma Android, verrà in seguito utilizzato come input alla parte di codice che si occupa della visualizzazione dell'immagine sull'interfaccia utente.

Capitolo 4
Implementazione dell'applicazione

Nei capitoli precedenti è stato analizzato in dettaglio l'algoritmo KinectFusion, sono stati descritti gli aspetti fondamentali della programmazione dei chip grafici NVIDIA, il linguaggio CUDA su cui tale programmazione è basata ed infine sono state illustrate le librerie impiegate nella realizzazione di questo progetto e il processo che ha permesso la compilazione del codice di KinectFusion su piattaforma Android.

Per consentire a tutte le tecnologie hardware e software finora illustrate di coesistere e di interagire fra di esse, nonché permettere l'effettiva usabilità dell'algoritmo KinectFusion in ambiente mobile, è stata sviluppata un'applicazione Android in grado di fare cooperare in maniera fruibile tutte le componenti in gioco mantenendo un'interfaccia utente semplice e intuitiva.

In questo capitolo vedremo i maggiori dettagli implementativi dell'applicazione realizzata, sia in termini dell'interfaccia grafica dell'applicazione, sia per quanto concerne l'interfacciamento con l'algoritmo KinectFusion e con il sensore utilizzato.

4.1 L’interfaccia utente

Quando un utente avvia una qualsiasi applicazione, l’interfaccia è sempre l’elemento che lo accoglie all’interno della stessa. Per questo motivo, è buona norma che tale interfaccia rimanga semplice e intuitiva, al fine di rendere agevole l’esperienza dell’utente nell’utilizzo di tale applicazione.

Un’interfaccia semplice e minimale non favorisce soltanto l’approccio dell’utente all’applicazione ma agevola anche il lavoro del programmatore, che in fase di implementazione del software ha la necessità di eseguire un numero considerevole di volte tale applicazione, al fine di effettuare il debugging delle modifiche appena apportate. Contemporaneamente vi è la necessità, da parte dell’utente, di ottenere un accesso immediato alle funzionalità che l’applicazione deve fornire, per evitare confusione nel suo utilizzo.

Ci concentriamo adesso a presentare l’aspetto puramente grafico degli elementi di cui è composta l’interfaccia utente dell’applicazione sviluppata, rimandando la descrizione specifica dell’implementazione degli stessi al paragrafo successivo.

Per una migliore fruibilità delle funzionalità dell’applicazione, l’interfaccia è stata implementata per essere visualizzata in modalità *landscape*, ovvero con il lato più lungo del dispositivo posizionato orizzontalmente.

All’avvio dell’applicazione l’interfaccia che si presenta all’utente è quella in **Figura 18**.

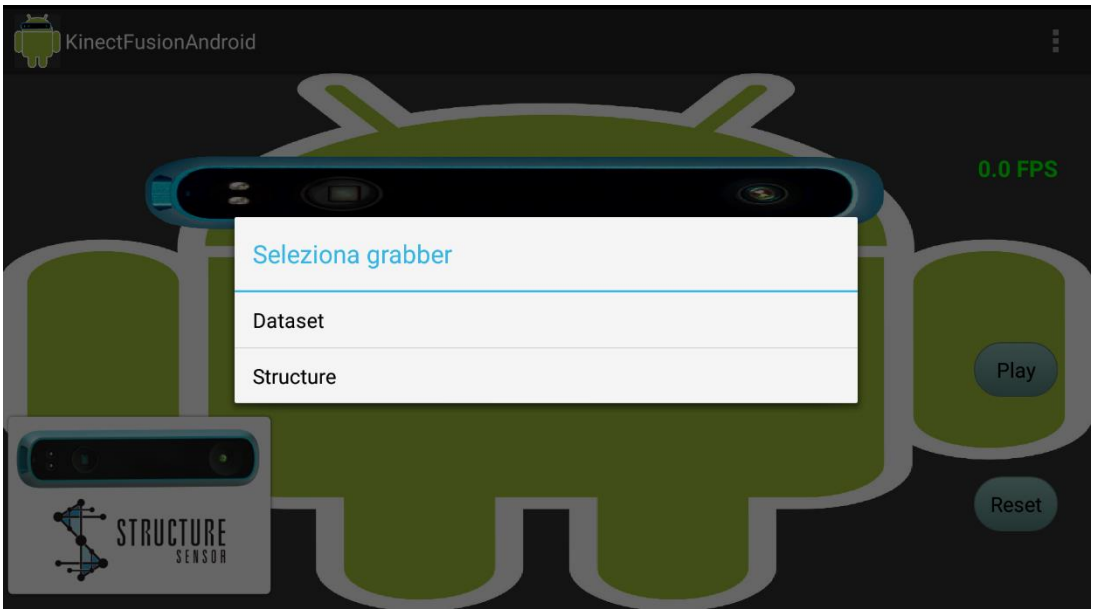

Figura 18: Interfaccia all’avvio dell’applicazione

Come è stato spiegato nel Capitolo 3, relativamente all'InputOutputLibray, l'applicazione permette l'acquisizione dell'immagine da due fonti: uno specifico dataset ed il sensore Structure. All'avvio dell'applicazione, è quindi predisposto un dialog per la scelta della sorgente.

Una volta scelto il tipo di grabber che si desidera utilizzare e se l'inizializzazione è andata a buon fine, un apposito messaggio di notifica segnala la corretta creazione del grabber.

In **Figura 19** è possibile vedere l'interfaccia principale dell'applicazione una volta che è stato scelto e creato il grabber. Nella figura sono stati cerchiati in rosso gli elementi principali dell'applicazione.

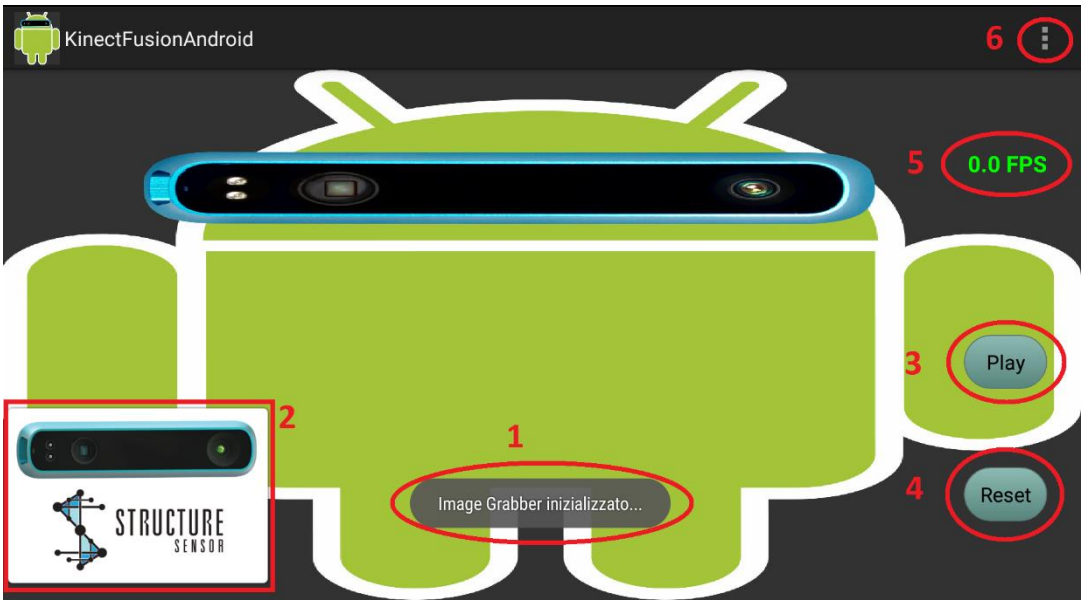

Figura 19: L'interfaccia principale dell'applicazione

Per prima cosa notiamo il messaggio di notifica di cui abbiamo appena parlato (numero 1 in **Figura 19**), che ci informa della corretta creazione del grabber.

L'elemento numero 2, in basso a sinistra dell'interfaccia, è il riquadro all'interno del quale verrà visualizzata la depth map proveniente dal dataset o dallo Structure, input dell'algoritmo KinectFusion.

Gli elementi denotati con 3 e 4 sono i pulsanti con i quali si può controllare l'avvio, la pausa, la ripresa ed il reset dell'algoritmo. Più nello specifico, l'elemento evidenziato con il numero 3 raffigurante la scritta «Play» rappresenta

il pulsante per l'avvio dell'algoritmo. Una volta che l'algoritmo è in funzione, la scritta all'interno di tale pulsante diventerà «Pause», cambiandone di fatto la semantica e l'utilità. Nel caso in cui il pulsante «Pause» venga selezionato, l'algoritmo verrà messo in pausa e la scritta sul pulsante diverrà «Resume». Da ora in avanti, sul pulsante si alterneranno ad ogni pressione le scritte «Pause» e «Resume». Tale pulsante otterrà nuovamente la funzione di «Play» solo tramite il reset dell'algoritmo o il riavvio dell'applicazione.

L'elemento 4, come intuibile dalla scritta riportata al suo interno, è invece il pulsante «Reset». Tale pulsante, al contrario del pulsante di Play/Pause/Resume, non cambia la sua semantica durante il ciclo di vita dell'applicazione e permette all'utente di azzerare il lavoro fatto da KinectFusion fino a quel momento. Se l'utente desiderasse in seguito avviare una nuova ricostruzione dovrà premere nuovamente il pulsante Play.

Per quanto riguarda il comportamento dell'interfaccia, alla pressione del tasto di Reset è sicuramente utile evidenziare la comparsa di un altro dialog tramite il quale l'utente può decidere di confermare il reset dell'algoritmo o annullare tale scelta.

Continuando a descrivere i componenti dell'interfaccia evidenziati in **Figura 19**, arriviamo al numero 5. Tale elemento non è altro che una casella di testo che, durante l'esecuzione dell'algoritmo, permetterà la visualizzazione del numero di FPS (frame al secondo), che l'algoritmo KinectFusion riesce ad elaborare.

Il numero 6, invece, rappresenta la classica icona per i menu delle applicazioni Android, tramite la pressione di tale pulsante è possibile usufruire di un piccolo menu messo a disposizione per questa applicazione e che è possibile osservare in **Figura 20**.

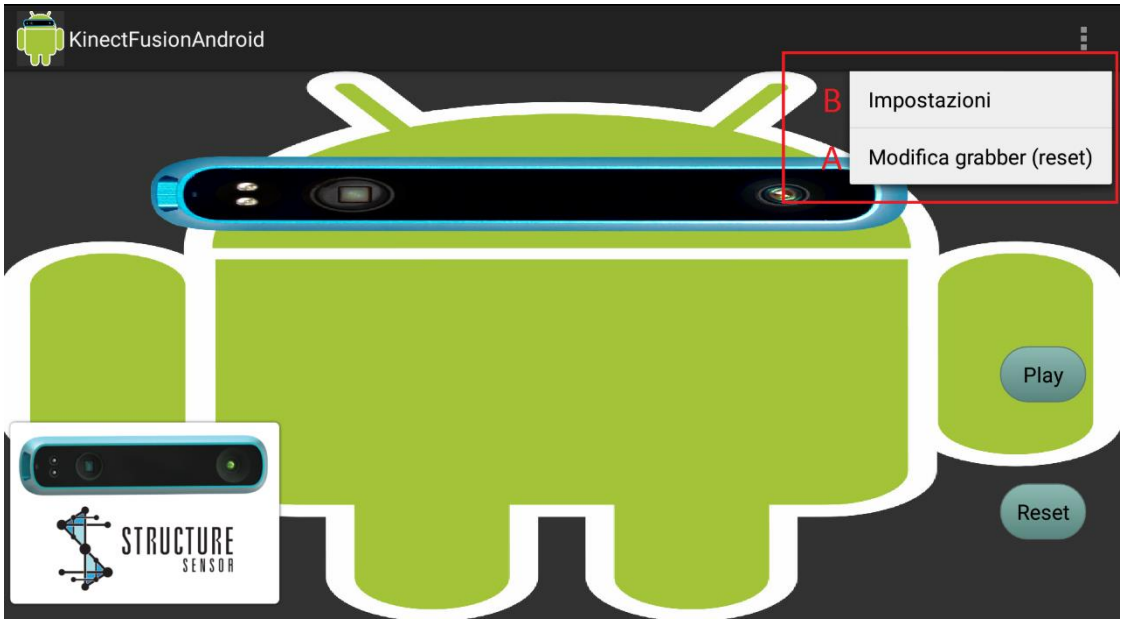

Figura 20: Menu dell'applicazione

Tale menu fornisce 2 scelte. Con la scelta A evidenziata in **Figura 20**, riportante la scritta «Modifica grabber (reset)», l'utente può cambiare il tipo di grabber da lui scelto in precedenza e quindi passare dal tipo dataset a quello Structure o viceversa. Tale switch prevede necessariamente l'inizializzazione del nuovo grabber e di conseguenza il reset del lavoro che si stava svolgendo in precedenza. La scelta B, invece, permette di accedere al menu di impostazioni, in cui è possibile modificare alcuni parametri per l'algoritmo KinectFusion. Vediamo questo menu nella **Figura 21**.

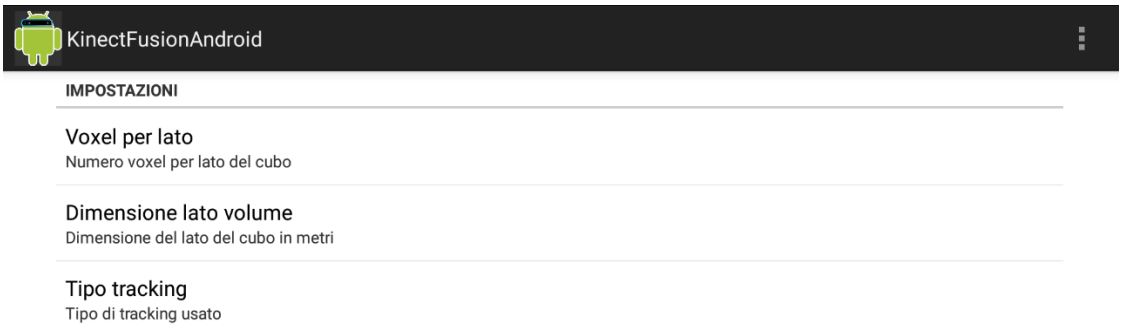

Figura 21: Impostazione dei parametri per KinectFusion

All’interno di questo menu, tramite le tre voci presenti, è possibile procedere all’impostazione di tre parametri di configurazione di KinectFusion.
La prima voce fornisce la possibilità di settare come parametro dell’algoritmo KinectFusion il numero di voxel per lato del volume tridimensionale, specificando un valore che sarà valido per tutti i tre lati. Selezionando questa opzione, appare un menu di scelta in cui sono riportati valori da 128 a 512 a intervalli di 32, come è possibile vedere in **Figura 22.**

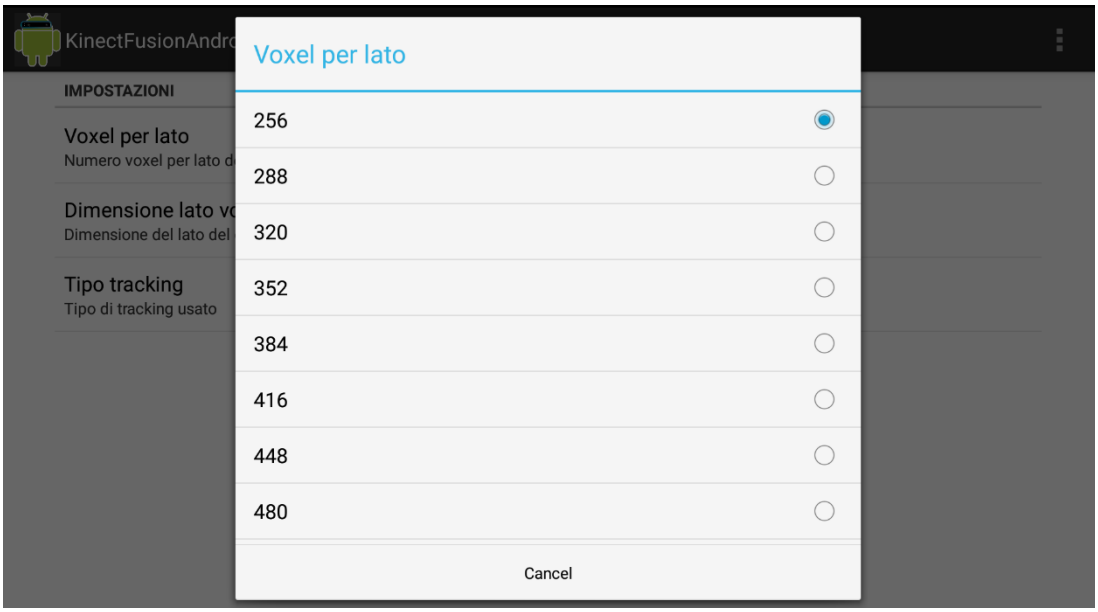

Figura 22: Scelta del numero di voxel per lato

La seconda voce in **Figura 21** permette invece all’utente di scegliere le dimensioni della scena che vuole riprendere. Il menu che appare alla pressione di questa voce è rappresentato da uno slider con un valore in metri raffigurato appena sopra di esso. Tramite questo slider l’utente può scegliere di rappresentare scene, all’interno di un cubo immaginario con lati che vanno da 10 cm a 3 metri, dipendentemente dall’ordine di misura di ciò di cui vuole ottenere la ricostruzione 3D. Lo step fra la selezione di un valore e quello successivo è di 10 cm. Lo slider è mostrato in **Figura 23**. Anche in questo caso la misura vale in modo uniforme per i tre lati del volume.

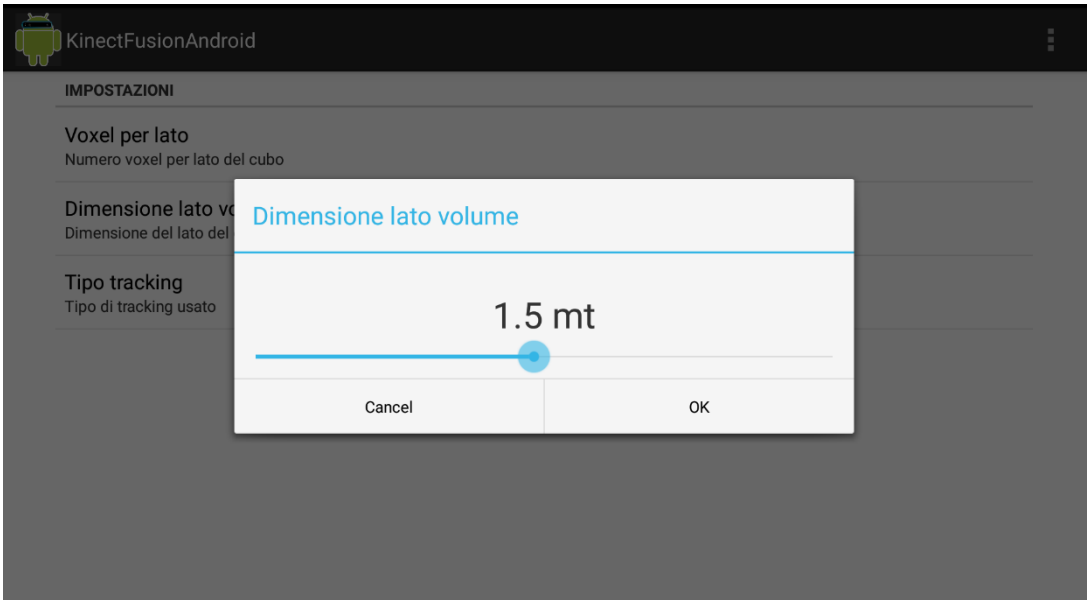

Figura 23: Slider per la scelta della dimensione della scena

Infine, la terza ed ultima voce del menu di impostazioni permette all'utente di selezionare il tipo di tracking che desidera utilizzare nell'esecuzione dell'algoritmo KinectFusion. Le scelte possibili, come visibile in **Figura 24**, ricadono tra «ICP» e «GEOMETRIC».

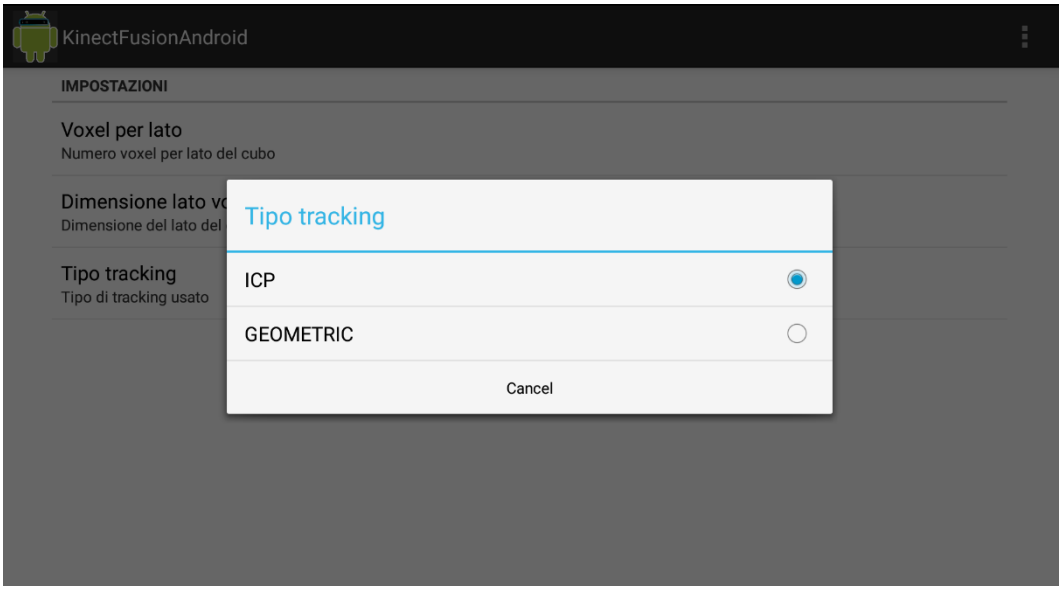

Figura 24: Scelta del tipo di tracking

Una volta effettuata una qualsiasi modifica da questi tre menu l'applicazione mostra un dialog che avvisa l'utente di dover riavviare l'applicazione per rendere effettive le modifiche appena effettuate.
Infine, lo sfondo dell'applicazione è utilizzato per la visualizzazione dei modelli 3D ricostruiti da KinectFusion. Possiamo vedere un esempio in **Figura 25**.

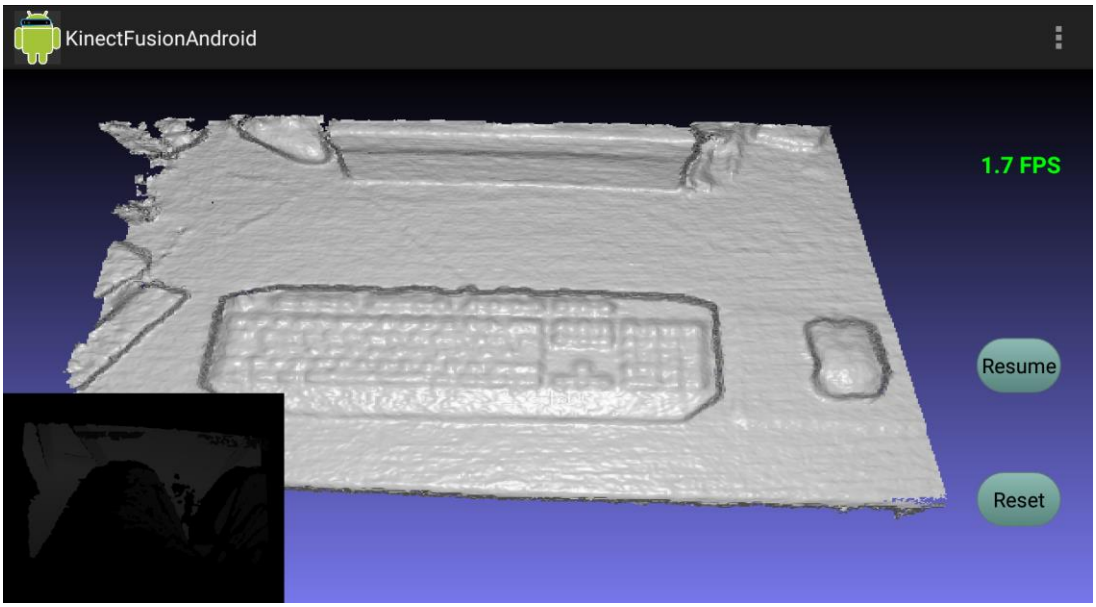

Figura 25: Sfondo dell'applicazione usato per visualizzare la ricostruzione

4.2 Dettagli implementativi

Abbiamo fin qui descritto la parte grafica dell'applicazione, dando molti spunti sui vari componenti software che fanno parte della stessa. Leggendo il paragrafo precedente potremmo ora domandarci: cosa permette di scegliere il dataset dalla memoria del tablet? Oppure, come vengono gestiti i pulsanti Play/Pause/Resume e Reset? O ancora, come viene visualizzata l'immagine di ricostruzione 3D sullo sfondo?
Nel corso di questo paragrafo cercheremo di dare una risposta a queste domande, tenendo inoltre in considerazione tutti gli aspetti software che non sono direttamente percepibili dall'interfaccia utente. Alcuni di questi potrebbero essere: come viene gestita l'operazione di collegamento/rimozione del sensore Structure? Oppure, come è gestito il ciclo iterativo di acquisizione

dell'immagine, esecuzione dell'algoritmo KinectFusion e stampa a video del risultato?

Si proverà in questo paragrafo a fare una panoramica generale e completa di tutti i vari aspetti software delle varie componenti che costituiscono l'applicazione sviluppata.

4.2.1 Implementazione dell'interfaccia grafica

A livello implementativo, l'interfaccia principale dell'applicazione (**Figura 19**) è definita nel file «activity_main.xml» e realizzata tramite l'utilizzo dell'editor grafico messo a disposizione da Android all'interno di Eclipse. Tale interfaccia è stata costruita sovrapponendo tre layout in maniera tale da diversificarne il contenuto e favorire il posizionamento in modo corretto sullo schermo degli oggetti in essi contenuti, a seconda della risoluzione del dispositivo utilizzato. Vediamo l'albero rappresentante la distribuzione dei layout in **Figura 26**.

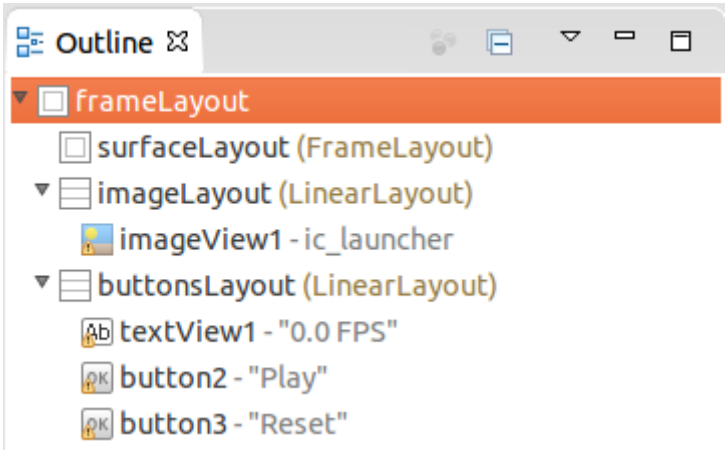

Figura 26: Albero dei layout dell'applicazione

Come è possibile vedere dalla figura l'interfaccia è composta da un layout principale, il quale racchiude tre layout sovrapposti (deducibile dal fatto che si trovano allo stesso livello dell'albero). L'ordine di sovrapposizione è dato dalla sequenza in cui i layout sono dichiarati nell'albero. Nel nostro caso il primo ad essere inserito, il quale quindi si ritroverà al di sotto degli altri, è il componente chiamato *surfaceLayout*. Su questo layout verrà disegnata l'immagine della

ricostruzione 3D. Per adempiere a questo compito, all'avvio dell'applicazione viene agganciata a questo layout una SurfaceView gestita tramite OpenGL.

Al di sopra del *surfaceLayout* è posizionato quello che è stato chiamato *imageLayout*. Tale layout ha la sola funzione di contenere l'ImageView nella quale sarà visualizzata la sequenza di immagini depth e di permetterne la giusta collocazione nell'angolo in basso a sinistra.

Infine, al di sopra dei due precedenti layout è situato il *buttonsLayout*, il quale ha il compito di contenere i pulsanti dell'applicazione e la TextView con la quale viene presentato a schermo il valore di FPS, favorendone la giusta collocazione sulla parte destra dello schermo e la corretta distanza tra gli stessi. Il posizionamento di questo layout al disopra degli altri permette ai due pulsanti di essere correttamente utilizzati.

Come è stato anticipato nel paragrafo precedente, l'interfaccia dell'applicazione è stata predisposta per essere visualizzata sempre in modalità landscape. Per fare ciò è necessario modificare il file «AndroidManifest.xml» del progetto, il quale descrive i componenti principali di una applicazione Android, come ad esempio i permessi richiesti e i diversi componenti grafici presenti. In questo file, sotto il tag <*activity*> indicante l'interfaccia grafica principale, è stato aggiunto l'attributo *android:screenOrientation="landscape"*. La specifica di tale attributo permette all'applicazione di restare sempre in modalità landscape anche nel caso in cui il dispositivo venisse ruotato.

4.2.2 Rilevamento di una connessione/disconnessione dello Structure dalla porta USB

Nel caso in cui si selezioni dal dialog iniziale (**Figura 18**) il sensore Structure come input dell'algoritmo, al fine di permetterne il corretto funzionamento sul tablet si sono dovuti considerare vari aspetti.

Il primo tra questi è quello dovuto ad una connessione o disconnessione del sensore dalla porta USB del dispositivo mobile. Nel parlare di connessione del sensore al dispositivo, si deve inoltre scindere due casi. Il primo è l'eventualità che il sensore sia già collegato alla porta USB all'avvio dell'applicazione, il

secondo è il caso in cui il sensore venga collegato alla porta USB ad applicazione già avviata. Nel primo caso, all'avvio dell'applicazione bisogna verificare la presenza di dispositivi collegati alla porta USB. Per fare ciò, si è fatto uso di un componente chiamato *UsbManager*, messo a disposizione dalla piattaforma Android. Tramite tale componente è possibile ottenere una lista dei dispositivi collegati alla porta USB. Successivamente si itera su questa lista controllando se uno dei dispositivi USB collegati riporta il vendor e product id attesi (ovvero quelli dello Structure che stiamo utilizzando), se questo controllo ha esito positivo lo Structure è stato rilevato e, tramite l'utilizzo dell'*UsbManager*, viene richiesto all'utente di concedere il permesso all'applicazione per utilizzare il dispositivo, il quale in caso positivo consentirà di poter finalmente avviare il sensore Structure e permettere l'esecuzione dell'algoritmo KinectFusion. La richiesta di permessi vale anche se lo Structure viene collegato ad applicazione già avviata ed è visibile in **Figura 27**.

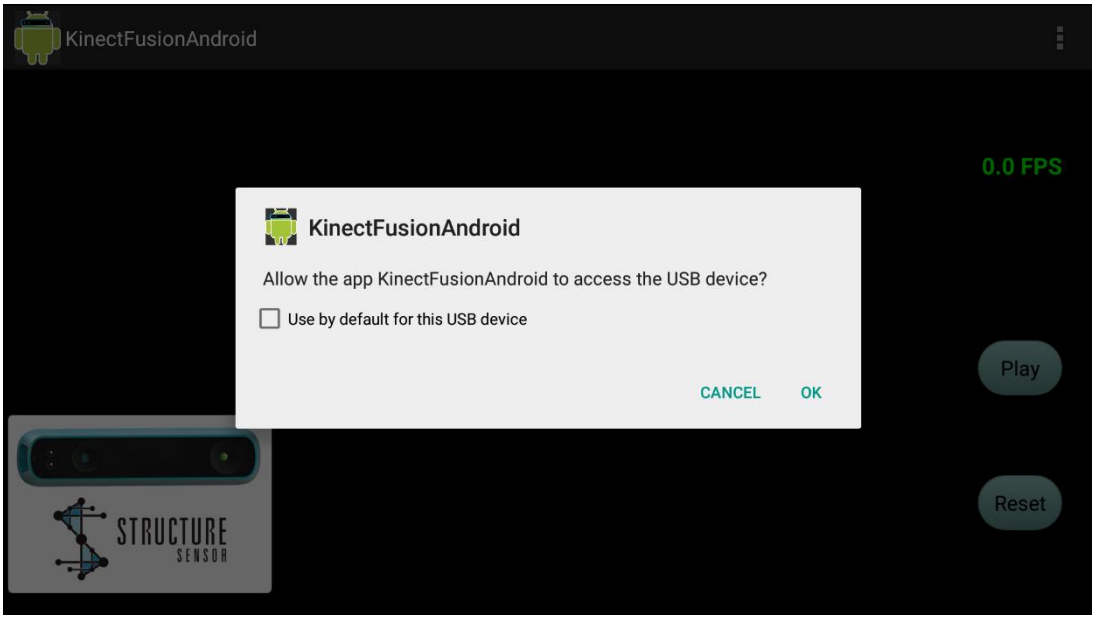

Figura 27: Richiesta dei permessi per utilizzare lo Structure

E' importante notare che questa operazione è disponibile soltanto per i dispositivi mobili dotati della funzionalità USB Host, ovvero in grado di comandare eventuali dispositivi connessi alle porte USB. In assenza di tale funzionalità non

vi sarà modo di comunicare con il sensore Structure, rendendone di fatto impossibile l'utilizzo.

Come abbiamo già detto, lo Structure può essere collegato o scollegato ad applicazione già avviata. Per questa evenienza all'avvio dell'applicazione viene anche registrato un *BroadcastReceiver*, altro componente disponibile sulla piattaforma Android che permette di mettersi in ascolto di specifici eventi. In questo caso questo componente verrà registrato per controllare le azioni di *ATTACHED* e *DETACHED* di un generico dispositivo USB. Tale receiver resta perciò costantemente in attesa di un possibile evento di connessione/disconnessione sulla porta USB. Quando un dispositivo USB viene collegato, il receiver cattura l'evento e controlla le informazioni del dispositivo USB connesso. Anche in questo caso, dopo aver controllato i vendor e product id del dispositivo e quindi aver riconosciuto lo Structure, si procede alla richiesta dei permessi come già visto in precedenza. Se invece il receiver cattura un evento di disconnessione, magari nel frattempo che l'algoritmo KinectFusion è in funzione, tutte le operazioni dell'applicazione vengono messe in pausa, viene mostrato un messaggio di allerta e successivamente l'applicazione viene resettata mostrando il dialog di scelta del grabber visto nel paragrafo precedente.

Nel gestire le funzionalità che riguardano lo Structure è necessario che l'intera applicazione abbia modo di verificare l'effettiva presenza del dispositivo alla porta USB e se i permessi necessari siano stati effettivamente concessi. Per questi motivi, sono state dichiarate due variabili booleane globali chiamate *structureAttached* e *structurePermission* con la semantica che è facilmente intuibile dal nome. Lo stato di tali variabili viene modificato all'interno del receiver. All'avvio dell'applicazione queste variabili sono impostate a *false* di default, quando lo Structure viene collegato *structureAttached* viene settata a *true* e se i permessi vengono successivamente concessi anche *structurePermission* viene settato a *true*. Un controllo su queste variabili viene fatto nell'applicazione al momento di lanciare l'algoritmo KinectFusion e di conseguenza iniziare ad acquisire le immagini dal dispositivo. Infine, nel caso in cui il dispositivo venisse

improvvisamente rimosso, il receiver che gestisce il relativo evento imposterà entrambe le variabili a *false*.

4.2.3 Scelta del dataset

Abbiamo visto in **Figura 18**, all'inizio di questo capitolo, il dialog per la scelta del tipo di grabber che si presenta all'utente all'avvio dell'applicazione. Nel caso in cui l'utente scelga di acquisire le immagini dal dataset, si deve fornire la possibilità di navigare all'interno del filesystem del tablet, al fine di scegliere il file di descrizione del dataset, il quale nel nostro caso specifico contiene i riferimenti alle coppie di immagini depth e rgb del dataset, nonché i loro timestamp. Per fare ciò alla pressione dell'opzione «Dataset» sul dialog è stato predisposto un pezzo di codice per l'apertura di un *FileChooser*, come indicato in **Figura 28**, tramite il quale l'utente può navigare tra le directory del tablet fino a selezionare il file di interesse.

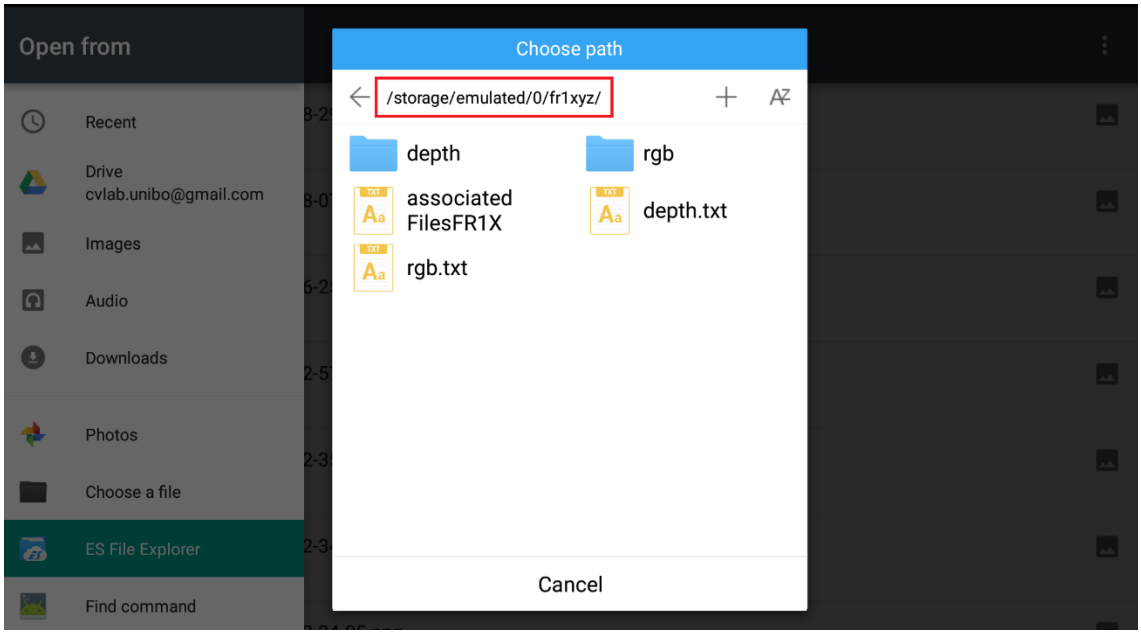

Figura 28: FileChooser per la scelta del dataset

Una volta selezionato il file corretto, viene chiamata la funzione che si occupa della creazione dell'*ImageGrabber*, passando come parametro il path del file che

è stato scelto (evidenziato dal riquadro in rosso nella **Figura 28**), concatenato al suo nome.

4.2.4 Modalità dell'applicazione

Nel momento in cui l'utente sceglie quale tipo di grabber intende utilizzare, l'applicazione entra in modalità *DATASET* o *STRUCTURE* dipendentemente dal grabber scelto. Per tenere traccia della modalità è stata istanziata una variabile *mode* a cui è assegnato uno dei due valori (*DATASET* o *STRUCTURE*).

Tale variabile è necessaria al fine di determinare il comportamento dell'applicazione alla pressione del pulsante «Play». Infatti, per esempio, nel caso in cui la modalità selezionata sia *STRUCTURE*, alla pressione di tale pulsante dovrà essere effettuato l'ulteriore controllo che il dispositivo sia effettivamente collegato e che l'applicazione abbia ottenuto i permessi necessari, cosa che non va fatta nel caso della modalità *DATASET*.

4.2.5 L'AsyncTask

In generale, in ambiente Android, l'*AsyncTask* è un thread asincrono capace di svolgere delle operazioni in background senza influire sul funzionamento dell'interfaccia grafica principale. Questo oggetto fornisce due metodi fondamentali: *doInBackground()* e *onProgressUpdate()*. Il primo è predisposto per contenere tutto il codice che deve essere eseguito dal thread asincrono. Tramite il secondo, invece, è possibile aggiornare gli elementi dell'interfaccia grafica che non sarebbero in altro modo accessibili dall'AsyncTask.

Nell'applicazione sviluppata per questa tesi è stato fatto uso di questo elemento, che rappresenta di fatto l'anima e il motore dell'applicazione. Infatti, oltre a operazioni di utility come gestione delle bitmap e conversioni delle immagini nei formati corretti, l'AsyncTask sviluppato si occupa soprattutto di acquisire le immagini tramite il grabber, lanciare l'esecuzione dell'algoritmo KinectFusion ed aggiornare sull'interfaccia grafica sia l'immagine contenente la depth map, sia quella della ricostruzione 3D che fa da sfondo all'interfaccia.

Per creare questo thread asincrono è stato necessario definire una nuova classe, chiamata *ImageGrabAsyncTask*, la quale estende *AsyncTask<SensorGrabberBGRD,Bitmap,Void>*, i parametri passati nella classe indicano rispettivamente l'oggetto che si desidera passare al metodo *doInBackground()* (nel nostro caso il generico grabber), l'oggetto che si intende passare al metodo *onProgressUpdate()* (nel nostro caso la bitmap contenente l'immagine depth che deve essere visualizzata) e per ultimo il parametro che si vuole restituire al termine dell'esecuzione del metodo *doInBackground()* (non siamo interessati a nessun valore di ritorno, quindi impostiamo *Void*).

Il comportamento che abbiamo voluto ottenere dall'applicazione alla pressione del pulsante «Play» è che ad ogni iterazione si iniziassero ad acquisire immagini dal grabber, si elaborassero tali immagini tramite KinectFusion ed infine si aggiornasse l'immagine depth (in basso a sinistra sull'interfaccia) e l'immagine di ricostruzione sullo sfondo. In caso di pressione del tasto «Pause» il comportamento che si è voluto realizzare è che il grabber continuasse ad acquisire le immagini, che fosse messa in pausa solo l'esecuzione dell'algoritmo KinectFusion, continuando perciò ad aggiornare l'immagine depth sull'interfaccia.

Per ottenere questo comportamento, all'interno della funzione *doInBackground()* è stato inserito un ciclo while la quale condizione di uscita è governata da una opportuna variabile booleana (*terminate*) controllata dalla chiusura dell'interfaccia grafica, che corrisponde dalla volontà di terminare l'*AsyncTask*.

All'interno del ciclo while la prima operazione che viene svolta è l'acquisizione dell'immagine depth (ed eventualmente quella RGB). Alla prima iterazione del ciclo vengono inizializzate diverse variabili di utilità e vengono estratti i parametri gestiti tramite le impostazioni dell'applicazione (*dimensioneLato*, *voxelLato, tipoTracking*), i quali serviranno in seguito come input all'algoritmo KinectFusion per l'inizializzazione delle proprie strutture dati.

Due delle variabili booleane di stato più importanti per il ciclo di vita dell'AsyncTask sono *isPaused* e *reset*. La prima, assume valore *true* se l'algoritmo è in pausa, *false* atrimenti; la seconda assume per la maggior parte

del tempo il valore *false* e diviene *true* solo quando vi è una richiesta di reset dell'algoritmo, ritornando allo stato iniziale dopo che il reset è avvenuto.

Ad ogni iterazione, dopo aver ottenuto l'immagine depth dal grabber, viene effettuato un controllo sulle variabili *isPaused* e *reset*. Questo controllo può dare esito positivo in due casi: o l'applicazione non è in pausa, di conseguenza il valore di *isPaused* sarà *false*, oppure l'applicazione è in pausa (*isPaused* ha valore *true*) ma è stato richiesto il reset dell'algoritmo KinectFusion, ovvero *reset* ha il valore *true*. Una volta passato questo controllo, se era stato richiesto un reset, si esegue l'algoritmo KinectFusion con un'immagine rappresentata da una matrice con gli elementi tutti a 0. Tale operazione permette all'algoritmo KinectFusion di resettare il proprio volume (**Paragrafo 1.2.3**) e di restituire un'immagine di ricostruzione vuota. Invece, nel caso in cui non fosse stato richiesto il reset, si esegue una conversione di unità di misura dell'immagine (da metri in millimetri) per renderla compatibile con ciò che si aspetta in ingresso KinectFusion e successivamente si richiama una funzione JNI, passando l'immagine di depth sotto forma di array ed i vari parametri di configurazione di KinectFusion. Tale chiamata JNI, ritorna il puntatore ad un oggetto *Mat* di OpenCV contenente l'immagine della ricostruzione tridimensionale, il quale subisce in un primo momento un cambio di formato dei colori (tramite la funzione *cvtColor* di OpenCV) e successivamente viene mostrato a video (sullo sfondo) grazie ad un oggetto della libreria OpenGL. In ogni caso, a conclusione dell'iterazione di ogni ciclo while l'immagine depth viene aggiornata chiamando la funzione *publishProgress()* e passando come parametro la bitmap relativa. Tale chiamata invoca conseguentemente il metodo *onProgressUpdate()*, al quale è consentito agire direttamente sugli elementi dell'interfaccia grafica e che quindi può aggiornare il contenuto dell'ImageView con l'immagine depth corrente.

Un'altra operazione svolta all'interno del metodo *doInBackground()* è il calcolo degli FPS. Tale calcolo viene effettuato registrando i tempi di ogni chiamata JNI per l'esecuzione di un'iterazione dell'algoritmo KinectFusion. Il valore degli FPS viene aggiornato ogni cinque iterazioni come media dei tempi delle ultime cinque esecuzioni e salvato in una variabile globale. Tramite il metodo

onProgressUpdate(), visto poco sopra, si accede al relativo oggetto TextView dell'interfaccia grafica, modificandone opportunamente il valore per la visualizzazione su schermo.

4.2.6 OpenGL

Descriviamo in questo paragrafo il modo in cui l'immagine della ricostruzione tridimensionale, risultante da ogni iterazione dell'algoritmo KinectFusion viene visualizzata sull'interfaccia.

Come è stato detto nel **Paragrafo 4.2.1**, l'interfaccia è strutturata da diversi layout sovrapposti tra cui il *surfaceLayout*. Abbiamo anche detto che a tale layout viene agganciato un oggetto (che è stato chiamato *surfaceViewKinfu*) che è istanza di una particolare classe da noi definita, la quale sfrutta un'implementazione messa a disposizione dall'ambiente Android della specifica OpenGL.

OpenGL è una specifica per differenti linguaggi e multipiattaforma che si prepone il compito di elaborare grafica 2D e 3D. Il framework Android mette a disposizione un'implementazione di OpenGL attraverso una libreria derivante da tali specifiche. Nello specifico Android si basa su OpenGL ES, ovvero una versione di OpenGL appositamente pensata per dispositivi integrati. Tramite l'utilizzo di tale libreria al programmatore è fornita la possibilità di disegnare, colorare, applicare specifiche texture e animare oggetti 2D e 3D, nonché controllare tutta una serie di fattori come ad esempio le luci di una scena e le dinamiche di riflessione di queste luci sugli oggetti.

Nell'applicazione sviluppata, la classe creata al fine di sfruttare le funzionalità di OpenGL è stata chiamata *SurfaceViewKINFU*. Tale classe estende la classe *GLSurfaceView* della libreria OpenGL. All'interno è dichiarato globalmente un oggetto *Renderer* che viene inizializzato nel costruttore. Inoltre, nel costruttore sono settati una serie di parametri come ad esempio *setPreserveEGLContextOnPause(true)* che permette di non perdere il contesto OpenGL qualora l'applicazione venisse messa in pausa, *setRenderer(myRenderer)* per impostare il renderer di questa istanza di OpenGL,

e l'impostazione per un continuo rendering della scena tramite *setRendererMode(GLSurfaceView.RENDERMODE_CONTINUOUSLY).*

All'interno della classe *SurfaceViewKINFU* è stato inoltre implementato il metodo *loadTextureByBitmap(Bitmap)* il quale è utilizzato dall'AsyncTask (**Paragrafo 4.2.5**) per poter controllare il Renderer e applicare l'immagine della ricostruzione (effettuata da KinectFusion) come texture alla nostra scena OpenGL.

Nell'implementazione OpenGL il Renderer è quel componente che si occupa di creare la SurfaceView (che è possibile pensare come il contenitore della scena), definire le forme geometriche che devono essere rappresentate, definire il punto di vista (sistema di riferimento) dell'utente rispetto alla scena, gestire gli shader per il disegno delle figure, animare gli oggetti presenti applicando su di essi opportune matrici di roto-traslazione e disegnare tali oggetti sulla SurfaceView.

Una classe per poter svolgere le funzioni di Renderer deve implementare l'interfaccia *GLSurfaceView.Renderer* ed implementarne i metodi fondamentali che sono:

- *onSurfaceCreated()*: Viene chiamato alla creazione del Renderer e inizializza una serie di elementi come il colore dello sfondo, il punto di vista dell'utente rispetto alla scena e gli shader per il disegno degli oggetti.
- *onSurfaceChanged()*: Viene chiamato quando l'applicazione ritorna da uno stato di pausa oppure lo schermo passa da un orientamento ad un altro, consentendo ad OpenGL di riadattare le misure della SurfaceView alla nuova configurazione dello schermo.
- *onDrawFrame():* Rappresenta il metodo nel quale viene effettivamente disegnata la scena sulla SurfaceView.

Nella nostra implementazione della classe per il renderer è stata riproposta infine la funzione *loadTextureByBitmap(Bitmap)* che essendo in tale classe ha visibilità diretta degli oggetti della scena e può effettivamente applicare la texture.

Per il nostro scopo di semplice visualizzazione dell'immagine, sono stati quindi predisposti due triangoli i cui vertici coincidono con le estremità dello schermo

del dispositivo e che si uniscono formando un rettangolo che copre per intero lo sfondo della nostra interfaccia (**Figura 29**).

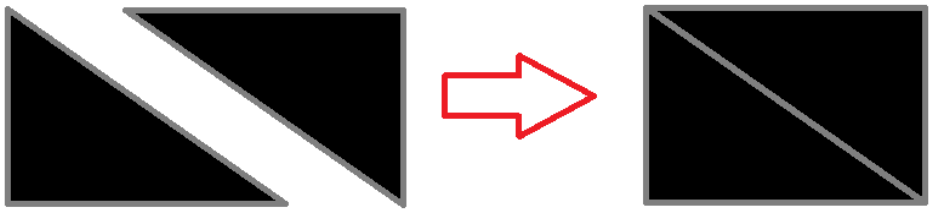

Figura 29: Rettangolo come unione di triangoli OpenGL

A tempo di esecuzione dell'applicazione verrà ad ogni ciclo dell'algoritmo KinectFusion applicata l'immagine delle ricostruzione come texture sui due triangoli.

4.2.7 Gestione delle impostazioni

L'ambiente Android mette a disposizione la classe *PreferenceManager* al fine di gestire (modificare e salvare) le impostazioni di alcuni valori che rimangono salvate per successivi utilizi dell'applicazione. Nel nostro caso i valori che vogliamo memorizzare tramite questa funzionalità: la quantità di voxel per lato del cubo, la distanza in metri che si desidera ricostruire ed il tipo di tracking.

Per la creazione del menu precedentemente presentato in **Figura 21**, è stata definita un'apposita classe chiamata *SettingFragment* la quale, per ottenere le funzionalità di menu, ha esteso la classe Android chiamata *PreferenceFragment*, facendo l'override dei metodi principali, ovvero *onCreate()* e *onSharedPreferenceChanged()*.

Il metodo *onCreate()* è quello che viene chiamato quando è richiesta l'apertura del menu delle impostazioni da parte dell'interfaccia grafica principale (tramite il menu visto precedentemente). Tale metodo carica le varie voci del menu da un file xml contenuto tra le risorse del progetto Android. All'interno di queste file xml sono definite, con gli appositi tag, tre categorie di preferenze. Come è possibile vedere nelle **Figure 22, 23** e **24** sono necessari tre diversi tipi di selettori per scegliere le impostazioni desiderate. Infatti, la **Figura 22** mostra una

lista di interi, la **Figura 23** mostra uno slider e la **Figura 24** una lista di stringhe. Per quanto riguarda l'ultimo tipo di menu (lista di stringhe) è stato sufficiente utilizzare il tipo *ListPreference* definito in Android. Negli altri due casi invece, è stato necessario definire delle classi apposite che sono state chiamate *IntegerListPreference* e *SeekBarPreference* le quali estendono entrambi *DialogPreference* permettendo di fatto di creare nuovi tipi di preferenze in base alle esigenze di sviluppo. E' interessante fare presente, che la classe *SeekBarPreference* implementa inoltre un listener chiamato *OnSeekBarChangeListener* il quale permette, ad ogni modifica apportata sullo slidebar, di aggiornare in tempo reale il valore in metri riportato sopra di essa. All'interno dei tag del file xml sono dichiarate varie componenti del menu come: le stringhe dei nomi delle preferenze, le stringhe delle loro descrizioni (le scritte più piccole in **Figura 21**) e i valori di default di tali preferenze.

Va inoltre sottolineato come, utilizzando questa metodologia di sviluppo, le varie impostazioni vengano effettivamente salvate sul dispositivo Android in maniera del tutto trasparente al programmatore, senza perciò richiedere la creazione di un'interfaccia grafica come nel caso della schermata principale dell'applicazione, né la gestione di particolari eventi di selezione delle voce o altro.

Come è già stato detto in precedenza, nella nostra applicazione al primo ciclo dell'AsyncTask vengono caricate le impostazioni, Ciò avviene tramite la classe *PreferenceManager* la quale, grazie al metodo statico *getDefaultSharedPreferences()* ritorna in un oggetto di tipo *SharedPreferences* la lista delle preferenze. Queste preferenze vengono alla fine estratte da tale oggetto riferendosi a loro con il nome ad esse assegnato nei tag xml descritti in precedenza.

Capitolo 5
Test e risultati

In questo capitolo verranno mostrate tre ricostruzioni tridimensionali di oggetti e scene di dimensioni crescenti, valutandone la qualità al variare del numero di voxel utilizzati per rappresentarli. Successivamente, verranno delineate le caratteristiche prestazionali ottenute dall'applicazione, riportando le tempistiche delle operazioni fondamentali di KinectFuson e le relative misure di FPS ottenute al variare delle impostazioni di voxel utilizzati e grandezza della scena ricostruita.

Tutti questi test sono stati effettuati utilizzando il tablet Shield della NVIDIA ed il sensore Structure della Occipital.

5.1 Valutazione qualitativa delle ricostruzioni

Per ogni oggetto, la misura della scena da ricostruire è stata scelta opportunamente in base alle dimensioni dell'oggetto stesso. Verrà quindi mostrata prima la normale immagine a colori dell'oggetto (per avere la percezione di ciò che si sta ricostruendo), e successivamente i modelli 3D ricostruiti, con numero di voxel per lato del cubo pari a 128, 256 e 416. Come impostazione minima è stata scelta 128 poiché rappresenta la misura sotto la quale una ricostruzione non sarebbe qualitativamente apprezzabile, invece è stato scelto come massimo 416 al fine di non saturare la memoria del tablet, infine 256 come valore (standard) intermedio.

Cominciamo quindi dall'oggetto più piccolo, valutandone la qualità delle ricostruzioni al variare della quantità di voxel utilizzati per rappresentarlo. Come primo oggetto è stato utilizzato un piccolo peluche rappresentante un pulcino (**Figura 30**).

Figura 30: Primo oggetto ricostruito - Peluche

Questo oggetto è caratterizzato da un ordine di misura intorno alla quindicina di centimetri, per questo motivo è stato utilizzato per tale oggetto un volume di

ricostruzione con un lato di 20 cm. Vediamo adesso, in ordine crescente del numero di voxel utilizzati, le ricostruzioni effettuate mediante KinectFusion su piattaforma Android.

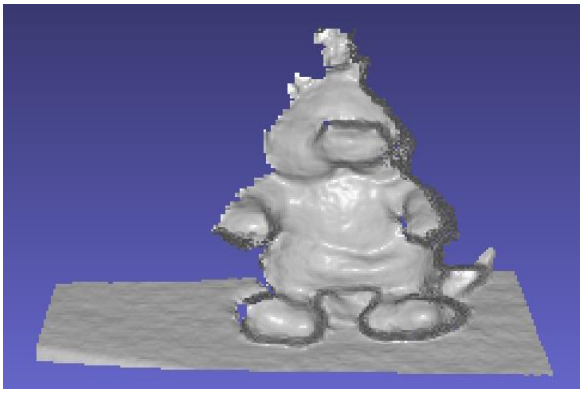

Figura 31: Peluche - 128 voxel

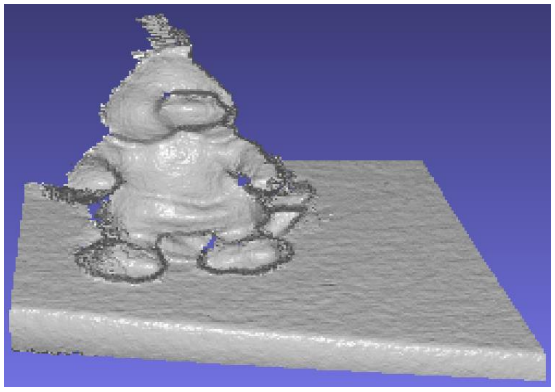

Figura 32: Peluche - 256 voxel

Come è possibile notare dalle **Figure 31** e **32**, cambiando in numero di voxel cambia la definizione dei dettagli della ricostruzione. Infatti, passando da 128 a 256 voxel per lato, notiamo un maggior dettaglio soprattutto nella zona tra la testa e il corpo, che appare marcatamente più divisa nella **Figura 32** piuttosto che nella **31**, nella quale invece le due zone sembrano quasi fuse. Inoltre, nella **32** è presente un maggior dettaglio del marsupio e delle zampe. Per questa ricostruzione non riportiamo l'esempio con 416 voxel poiché in tale ricostruzione non è stata riscontrata nessuna differenza evidente da quella con 256. Questo effetto è dovuto al fatto che in un volume di ricostruzione così piccolo la densità di voxel al suo interno è già elevata con l'impostazione da 256 voxel. Discuteremo più in dettaglio questo aspetto al termine di questo paragrafo.
L'oggetto di grandezza media scelto per la seconda ricostruzione è una chitarra, visibile in **Figura 33**.

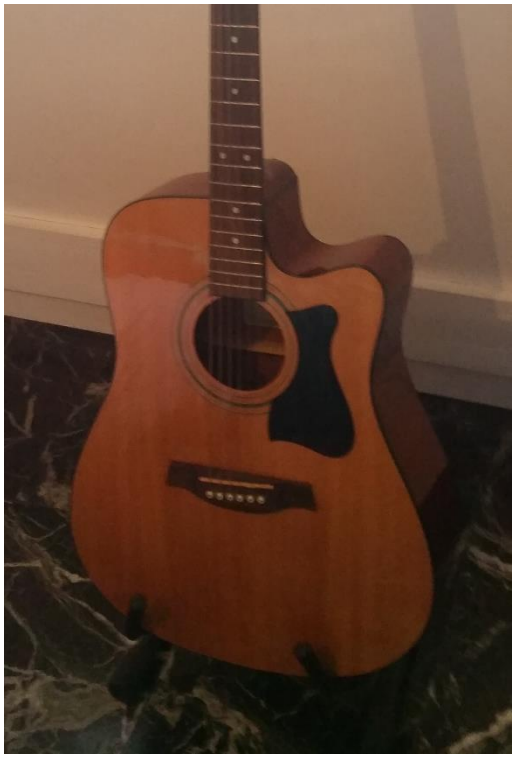

Figura 33: Secondo oggetto ricostruito - Chitarra

Per ricostruire l'oggetto di grandezza media è stato scelto un volume di un metro. Vediamo quindi le tre ricostruzioni al variare del numero di voxel.

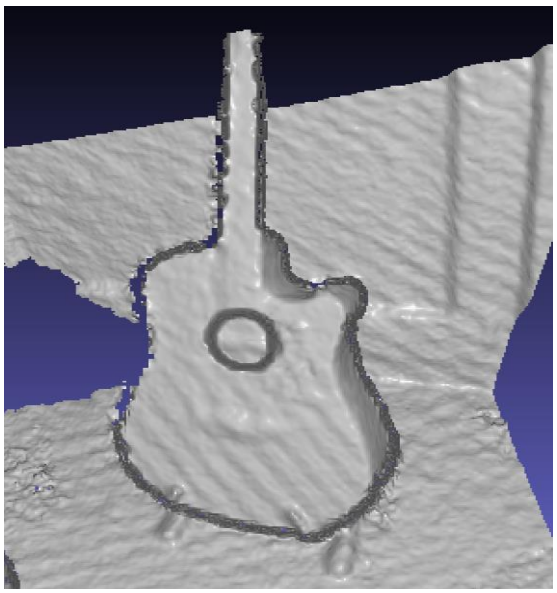

Figura 34: Chitarra - 128 voxel

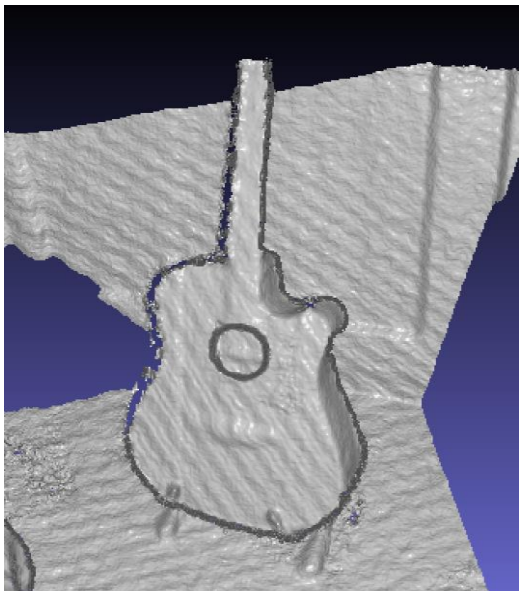

Figura 35: Chitarra - 256 voxel

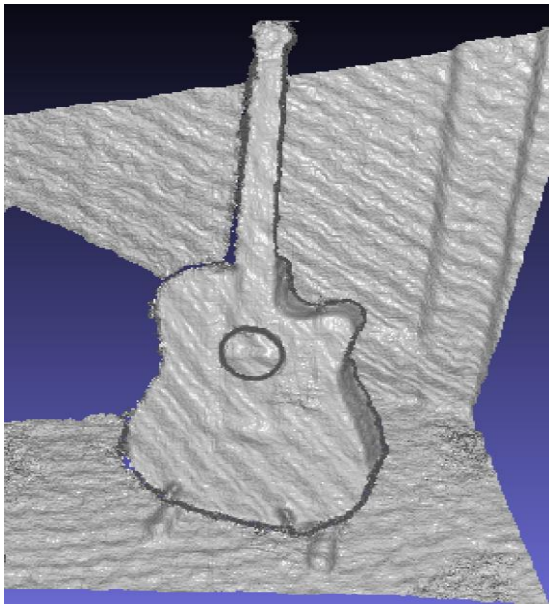

Figura 36: Chitarra - 416 voxel

Anche in questo caso è visibile tra le immagini da 128 e 256 voxel per lato un netto miglioramento generale dato dall'incremento dei voxel nel modello di ricostruzione. Andando nel particolare, in **Figura 35** è possibile notare lo spigolo destro della forma dalla chitarra risulta più definito rispetto alla **Figura 34**, anche il piedistallo che regge la chitarra è rappresentato in maniera più nitida. Inoltre, anche se non è elemento dell'oggetto che si intendeva ricostruire, è possibile vedere (sulla destra dell'immagine) una maggiore presenza degli spigoli della porta dietro la chitarra. In questo caso è stata riportata anche la ricostruzione con 416 voxel per lato ma possiamo notare che ancora non sono presenti differenze evidenti rispetto alla ricostruzione da 256 voxel per lato, la quale continua a soddisfare l'aspetto qualitativo. Vediamo con il prossimo esempio che questo non è sempre vero.

L'ultima scena che mostriamo rappresenta un tavolo con delle sedie. Sul tavolo è stata inoltre posizionata una pianta con lunghe foglie (per aggiungere ulteriori dettagli alla scena). L'immagine a colori della scena è mostrata nella **Figura 37**.

Figura 37: Terza scena ricostruita - Tavolo

Data l'ampiezza della scena, è stato impostato un volume di tre metri. Vediamo adesso in ordine le ricostruzioni effettuate evidenziando i dettagli caratteristici.

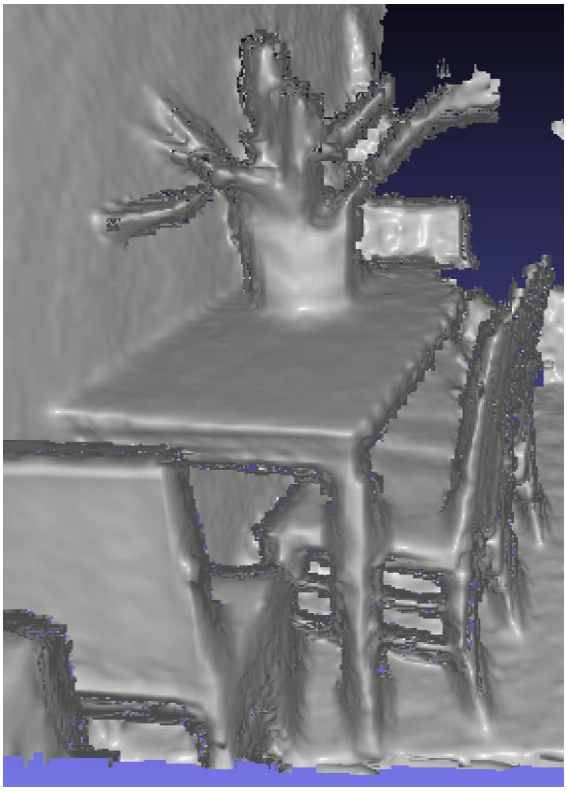

Figura 38: Tavolo - 128 voxel

Figura 39: Tavolo - 256 voxel

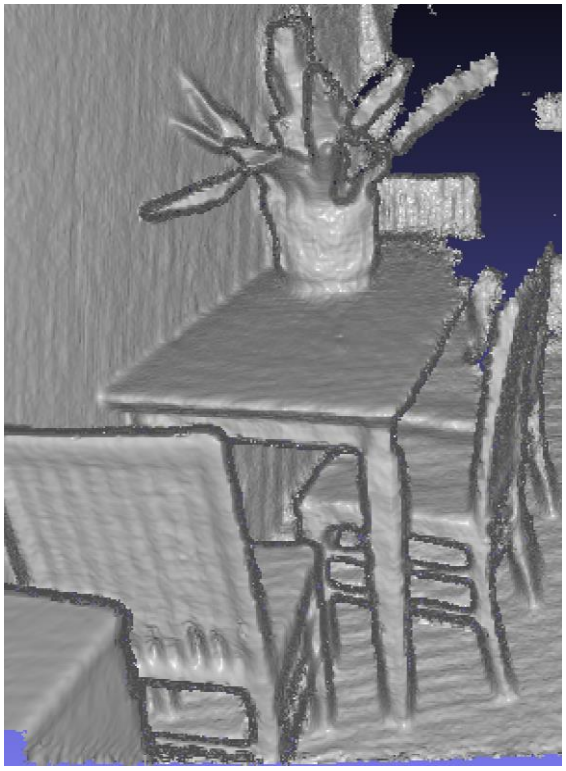

Figura 40: Tavolo - 416 voxel

Guardando la **Figura 38**, la prima osservazione che possiamo fare è che, al crescere della grandezza della scena che viene ricostruita, l'utilizzo di un basso numero di voxel potrebbe condurre ad una ricostruzione qualitativamente insufficiente. Infatti in tale immagine le spalliere delle sedie sembrano essere un blocco unico, sia il tavolo che la pianta sembrano uniti al muro ed il vaso al tavolo. Invece già dalla **Figura 39** si inizia a percepire la presenza delle bacchette nello schienale della sedia, si notano in maniera più delineata i poggiapiedi della sedia e anche la pianta e il vaso iniziano ad assumere maggiori dettagli. Inoltre, si inizia anche a percepire la presenza dello spigolo tra il sostegno (in legno) del tavolo ed il piano di marmo, che nell'immagine precedente non appariva quasi per nulla. Tale discorso vale in maniera ancora più netta nella **Figura 40**, nella quale si notano abbastanza distintamente le bacchette dello schienale della sedia, anche di quella più distante. I piedi (e poggiapiedi) delle varie sedie e del tavolo sono distinti in maniera netta, la pianta ha acquisito una propria fisionomia e lo spigolo tra il piano del tavolo e il suo sostegno in legno risulta adesso molto marcato.

In generale, come era attendibile, ad un aumento del numero dei voxel nel volume di ricostruzione corrisponde un incremento della qualità globale e dei dettagli delle ricostruzioni. Tale incremento della qualità, è maggiormente percepibile nelle scene più grandi. Per dare una spiegazione a questo fenomeno percettivo, intuitivamente possiamo pensare che nel rappresentare una piccola scena (es. 20cm), 128 voxel per lato possono fornire già una densità di voxel per centimetro di scena soddisfacente, tale per cui un ulteriore incremento di voxel risulterebbe per lo più superfluo. Invece, l'aumento del numero di voxel nella ricostruzione di una grande scena comporterebbe un guadagno nei dettagli maggiore. Proviamo a dimostrarlo osservando questi rapporti tra:

$$\frac{misura\ in\ cm\ del\ lato\ del\ volume}{numero\ di\ voxel\ per\ lato\ del\ volume}.$$

Con una distanza di 20cm si ha:

- $\frac{20\ cm}{128\ voxel} = 0.156\ cm/voxel$
- $\frac{20\ cm}{416\ voxel} = 0.048\ cm/voxel$

Con una differenza di $0.156 - 0.048 = 0.108\ cm/voxel$, di fatto una differenza minima tra le due ricostruzioni.

Invece, con una distanza di 300cm (3 m), si ha:

- $\frac{300\ cm}{128\ voxel} = 2.344\ cm/voxel$
- $\frac{300\ cm}{416\ voxel} = 0.721\ cm/voxel$

Con una differenza di $2.344 - 0.721 = 1.623\ cm/voxel$, che rappresenta un guadagno più consistente nella ricostruzione con 416 voxel rispetto a quella con 128.

5.2 Analisi delle performance

Per l'analisi delle prestazioni dell'algoritmo KinectFusion sul tablet Shield è stato utilizzato il noto dataset TUM [3], nello specifico la sequenza *fr1/xyz*. Le tempistiche riportate nelle tabelle che mostreremo, sono state acquisite direttamente tramite il codice C dell'algoritmo KinectFusion e riguardano esclusivamente il tempo effettivo impiegato dalle operazioni osservate, che sono:

il tracking della camera, l'integrazione volumetrica (che comprende il calcolo della TSDF) ed il raycasting. Invece, è importante sottolineare che il valore degli FPS mostrato sull'interfaccia è calcolato nel codice Java dell'AsyncTask e riguarda la tempistica della chiamata alla funzione JNI la quale comprende (oltre alle operazioni proprie di KinectFusion sopra menzionate), diverse operazioni di gestione delle strutture dati, upload e download delle immagini in KinectFusion; per questo motivo è stato riscontrato un ulteriore incremento di tempo per ogni frame in media di 50 millisecondi.

Nell'operazione di acquisizione dei dati prestazionali, è stata fissata la misura della ricostruzione a tre metri, al fine di elaborare sempre la stessa porzione di scena, data la natura del dataset. Sono state invece prese in considerazione le tre impostazioni di quantità di voxel per lato del volume proposte nel precedente paragrafo, ovvero: 128, 256 e 416. Le tre combinazioni che verranno esaminate sono quindi:

- 128 voxel & 3 metri
- 256 voxel & 3 metri
- 416 voxel & 3 metri

La misurazione del tempo di esecuzione delle diverse operazioni è stata effettuata sui primi 100 frame di utilizzo dell'algoritmo. I grafici che verranno mostrati riportano i tempi di esecuzione delle tre principali operazioni di KinectFusion (Track, Integrate e Raycast) in funzione del numero del frame elaborato, nonchè la media complessiva. Riportiamo in seguito i grafici per le varie combinazioni.

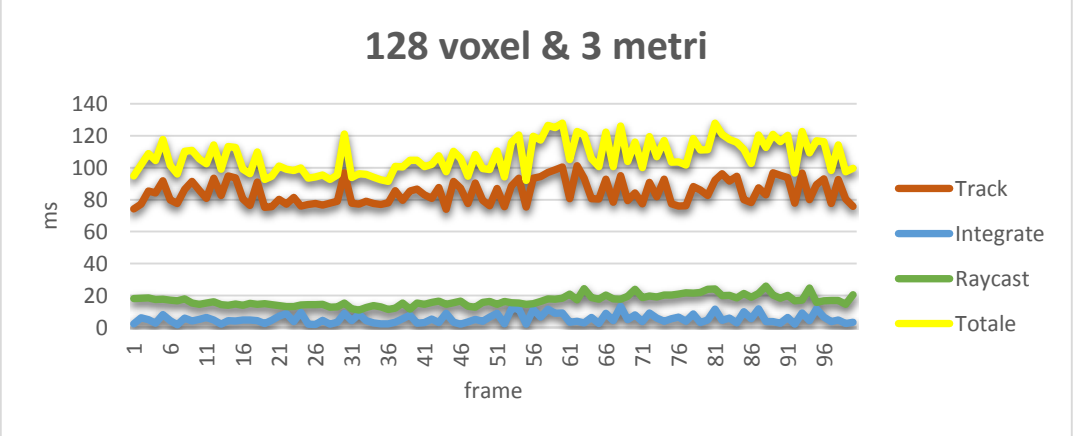

Figura 41: Tempi di esecuzione algoritmo (frame by frame) – 128 voxel

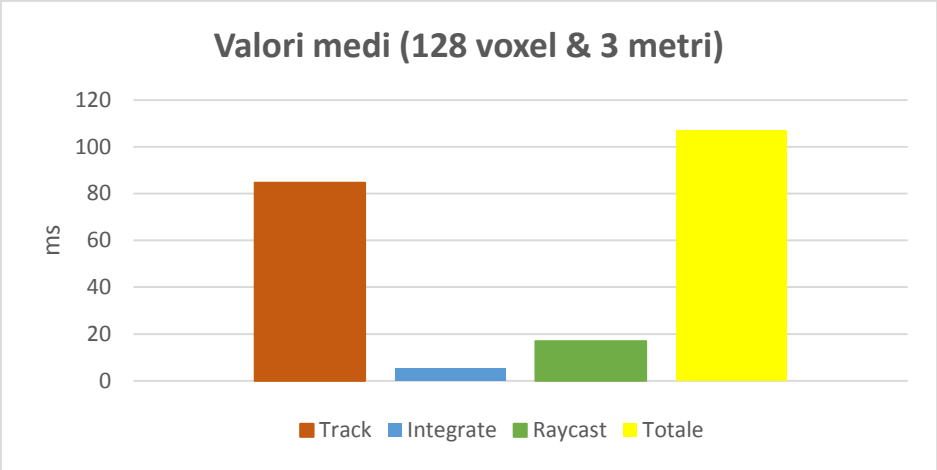

Figura 42: Tempi di esecuzione algoritmo (media) – 128 voxel

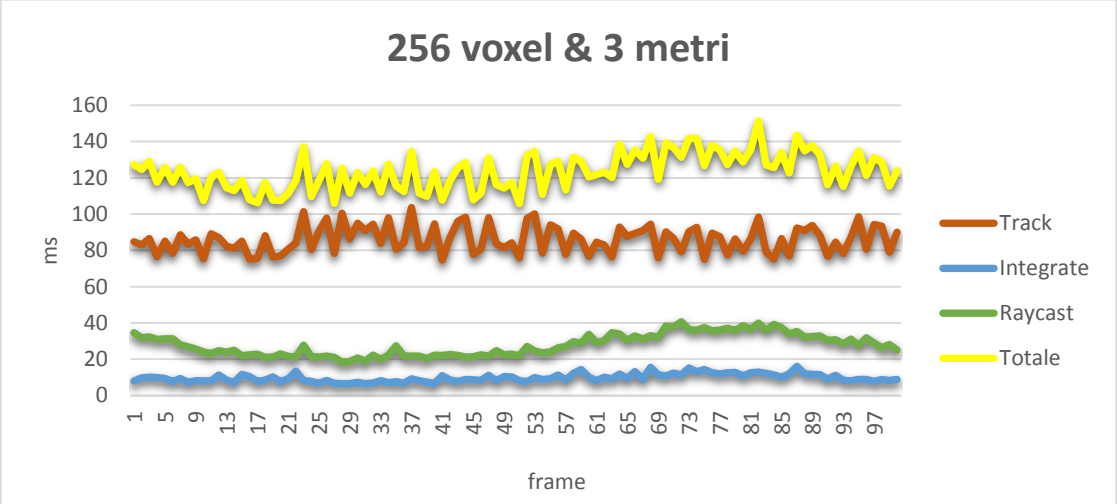

Figura 43: Tempi di esecuzione algoritmo (frame by frame) – 256 voxel

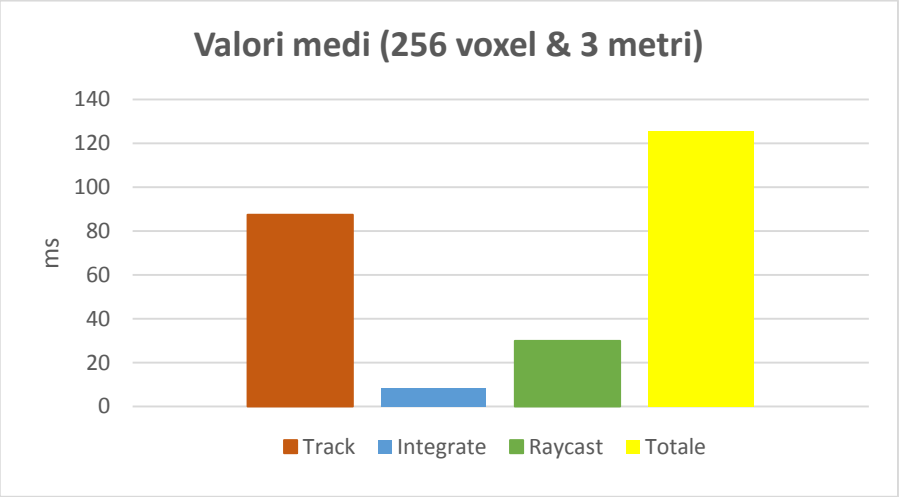

Figura 44: Tempi di esecuzione algoritmo (media) – 256 voxel

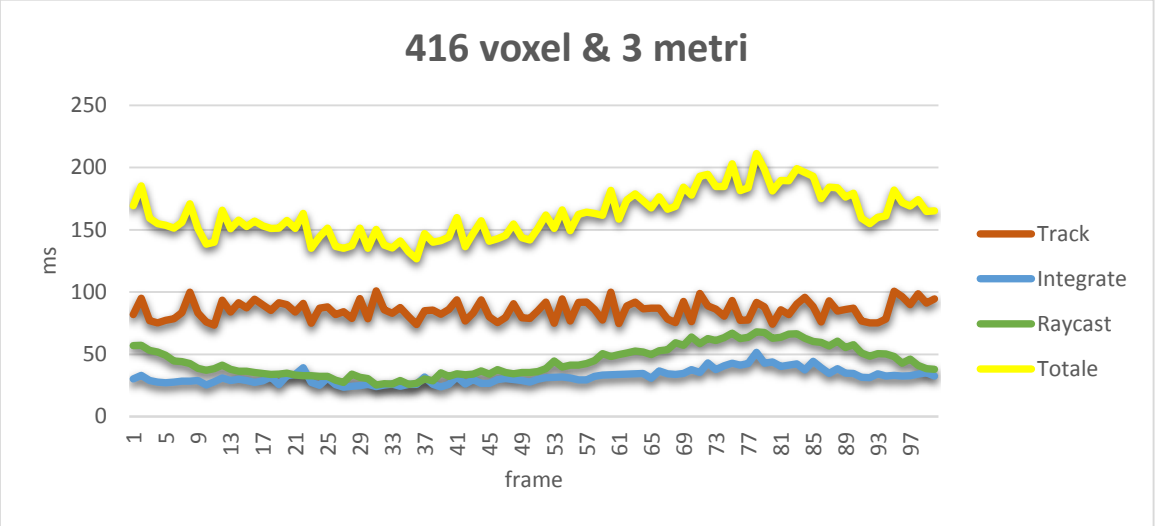

Figura 45: Tempi di esecuzione algoritmo (frame by frame) – 416 voxel

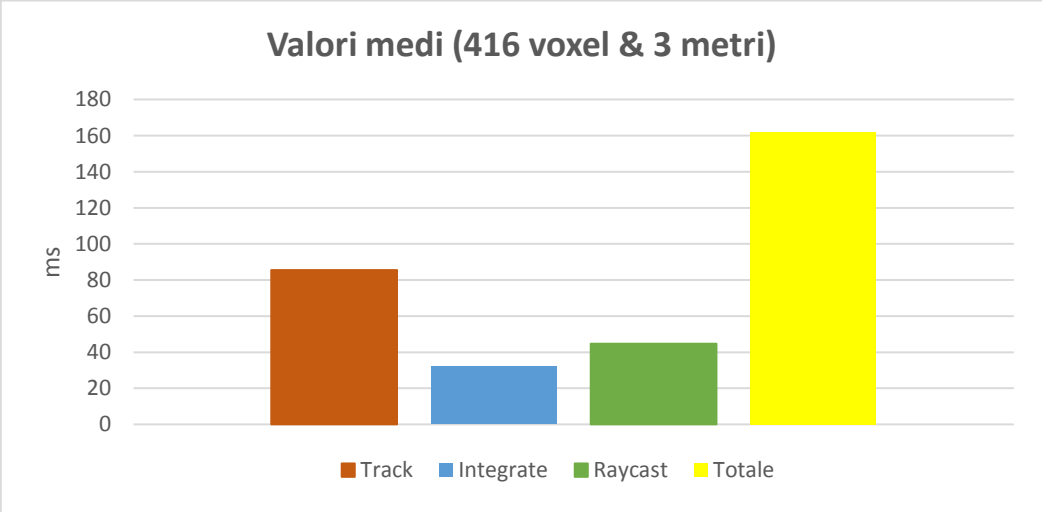

Figura 46: Tempi di esecuzione algoritmo (media) – 416 voxel

Iniziamo le valutazioni facendo delle osservazioni generali sui tre grafici principali.

In linea di principio, si nota che l'operazione che impiega il tempo maggiore è quella del tracking della telecamera, segue l'operazione di raycast e infine l'integrazione del volume.

Una crescita dei tempi di esecuzione tra i test con 128, 256 e 416 voxel per lato era sicuramente attesa ed è stata effettivamente riscontrata con questi rilevamenti. Infatti, osservando i valori medi delle somme dei tempi, si nota un incremento

delle tempistiche da 107 ms con l'impostazione da 128, a 125ms con quella da 256 e infine 162 ms con 416 voxel per lato. E' interessante notare che in tale incremento le tempistiche del tracking restano pressoché invariate nei tre grafici. Infatti le operazione per effettuare il tracking della telecamera, come spiegato nel Capitolo 1, si basano solo sul calcolo di mappe di vertici, di conseguenza non risentono del cambio del numero di voxel che prendono parte alla ricostruzione. Dell'incremento del numero di voxel ne risentono invece le operazioni di Raycast ed Integrate, le quali devono appunto operare sui voxel del volume. Il valore medio per l'operazione di integrazione, che comprende il calcolo della TSDF per ogni voxel, passa infatti attraverso le tre impostazioni da 5 ms a 8 ms fino a 32 ms. Mentre l'operazione di raycast passa da 17 ms a 30 ms fino a 44ms. Nei grafici, è possibile notare due particolari andamenti delle curve, il primo verso il basso intorno al frame 30, e il secondo verso l'alto intorno al frame 80, soprattutto nell'andamento del raycast. Intorno a tali frame, infatti, il dataset presenta un'inquadratura della scena rispettivamente più stretta e più larga, mostrata a colori in **Figura 47** e **48** rispettivamente.

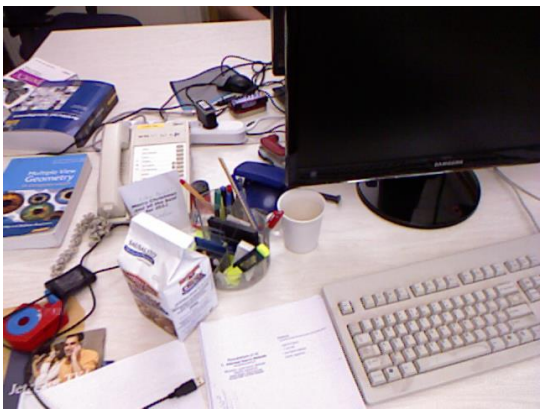

Figura 47: Scena stretta

Figura 48: Scena larga

L'algoritmo KinectFusion esegue il raycast solo sulla porzione di scena effettivamente inquadrata, per questo motivo elaborando una porzione di scena più stretta l'algoritmo si troverà a lavorare su meno voxel riducendo le tempistiche di esecuzione. Il discorso inverso vale nella scena più larga, nella

quale il numero dei voxel coinvolti dall'operazione di raycast è maggiore così come il tempo impiegato per elaborarli.

L'andamento verso il basso e verso l'alto è presente in maniera sempre maggiore al crescere del numero dei voxel usati per la ricostruzione.

Tutte le misurazioni effettuate si possono tradurre in termini di FPS (Frames per Second), che vengono effettivamente visualizzati dall'utente sull'interfaccia dell'applicazione. Per effettuare tale calcolo, partiamo dai valori medi delle somme dei tempi delle operazioni che abbiamo registrato, a cui dobbiamo sommare i 50ms (accennati in precedenza) dovuti al tempo di esecuzione delle operazioni accessorie.

La ricostruzione con 128 voxel ha un valore medio delle somme dei tempi di 106 ms a cui sommiamo i 50 ms delle operazioni accessorie ottenendo 156 ms per frame. Calcoliamo adesso la quantità di frame per secondo media che viene rilevata dall'applicazione: $\frac{1000\ ms}{156\ ms} = 6.4\ FPS$. Il valore degli FPS, in realtà oscilla in funzione del frame considerato, variando da 5.5 FPS minimi a 7.2 FPS massimi.

Passando dalla ricostruzione con 128 voxel a quelle con 256 voxel, abbiamo già evidenziato l'aumento del tempo impiegato dall'algoritmo KinectFusion. Da questo ne consegue una inevitabile diminuzione del valore di FPS. Infatti, per le ricostruzioni con 256 voxel è stato registrato un valore medio della somma pari a 125 ms, il valore di FPS che ne deriva è quindi di 5.7 FPS.

Considerando infine i 162 ms medi della somma per quanto riguarda i tempi della ricostruzione da 416 voxel per lato, il valore medio di FPS generato è di 4.7 FPS.

Come era facilmente attendibile il valore del numero di FPS decresce con il crescere del numero di voxel del volume e i relativi tempi di esecuzione.

A conclusione delle valutazioni fatte tra la qualità delle ricostruzioni e le tempistiche registrate, un buon compromesso è impostare il numero di voxel per lato a 256. Tale impostazione fornisce un valore di FPS medio intorno ai 5.7 FPS con una qualità variabile (come visto nel paragrafo precedente) al variare della

grandezza della scena, ma che risulta comunque sufficientemente dettagliata in un numero considerevole di scenari.

Conclusioni e sviluppi futuri

Nello sviluppo di questa tesi l'obiettivo primario di porting dell'algoritmo KinectFusion su piattaforma mobile, viste anche le ricostruzioni che sono state mostrate nei capitoli precedenti, ha avuto sicuramente un esito positivo. D'altra parte, in termini di usabilità, come è stato evidenziato dal numero di FPS generati, che anche nei casi più favorevoli si aggira intorno ai 7/8 FPS, non permette una acquisizione real-time del modello come di fatto avviene in un'esecuzione di KinectFusion in ambiente desktop. In questi termini, il collo di bottiglia è rappresentato dall'esiguo numero di core del processore grafico K1 del tablet Shield utilizzato (che ricordiamo essere di 192), rispetto al numero di core presenti su una scheda grafica per PC desktop, il quale si può attestare anche intorno al migliaio. Quindi è facilmente auspicabile che un incremento del numero di core sul dispositivo mobile porterebbe a migliori prestazioni nella fluidità e fruibilità dell'applicazione.

Dal punto di vista dell'applicazione Android, è stata creata un'interfaccia ai fini di rendere intuitivo e semplice l'utilizzo. Come ulteriore sviluppo si può pensare di aggiungere un riquadro per la visualizzazione dell'immagine RGB (a colori) ed altri pulsanti e schermate per ospitare nuove funzionalità, come ad esempio il salvataggio del modello 3D ottenuto oppure un pulsante per abilitare/disabilitare l'applicazione dei colori sul modello 3D ricostruito.

In termini delle dinamiche di acquisizione del modello, potrebbe essere utile inserire sull'immagine di depth un volume di grandezza modificabile in base alle dimensioni del volume utilizzato dall'algoritmo, colorando i punti della scena che risiedono spazialmente in esso per favorire la percezione all'utente di ciò che sta per ricostruire, permettendo di fatto la modellazione della sola porzione di scena desiderata e facilitando così un posizionamento iniziale del sensore.

Bibliografia

[1] Shahram Izadi et al., "Real-time 3D Reconstruction and Interaction Using a Moving Depth Camera," in *ACM*, 2011.

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

[3] J. Sturm, N. Engelhard, F. Endres, W. Burgard, and D. Cremers, "A Benchmark for the Evaluation of RGB-D SLAM Systems," in *IROS*, 2012.

[4] CUDA Overview. [Online]. http://www.cc.gatech.edu/~vetter/keeneland/tutorial-2011-04-14/02-cuda-overview.pdf

[5] CUDA C Programming Guide. [Online]. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html