Tesi di Laurea Magistrale

Tracking su dispositivo olografico di realtà aumentata

Localizzazione immersiva a sei gradi di libertà con camera monoculare

Riccardo Rodriguez

Relatore
prof. Andrea Sanna

Laurea Magistrale in Ingegneria Informatica

DAUIN
Politecnico di Torino
Torino, Piemonte
Dicembre 2023

Tracking su dispositivo olografico di realtà aumentata

Localizzazione in tempo reale a sei gradi di libertà con camera monoculare

Riccardo Rodriguez

Relatore
prof. Andrea Sanna Dauin

Sommario

Il posizionamento di oggetti tridimensionali nello spazio in due dimensioni definito da immagini o video costituisce uno dei campi di studio più prolifici della Computer Vision e suscita l'interesse del pubblico commerciale nonché di svariati settori industriali, da cui scaturiscono prospettive di crescita quasi esponenziale così come un'espansione continua dei casi d'uso. Oggi più che mai l'applicazione pratica delle conoscenze acquisite risulta di interesse generale con la disponibilità ad un pubblico sempre più vasto di software ed hardware dedicati alla realtà aumentata o Augmented Reality (AR), quasi ubique sui dispositivi digitali portatili più moderni, dagli smartphone a visori appositamente progettati, come ad esempio quello denominato Hololens, commercializzato da Microsoft.

Le applicazioni emergenti più promettenti della realtà aumentata fanno leva su un grado sempre crescente di interazione, grazie a prestazioni che sostengono precisione ed accuratezza adeguati pur mantenendo velocità che si accordano alle capacità percettive umane e non secondariamente ad interfacce sofisticate ma sempre meno intrusive.

Il presente lavoro si colloca nel suddetto contesto e parte dalla ricerca di una soluzione basata su algoritmi open source al problema specifico del tracking immersivo, con sei gradi di libertà, di oggetti affiancati da controparti appositamente modellate in CAD o software 3D apposito; l'intenzione dichiarata è che possa affiancarsi a soluzioni commerciali già esistenti, offrendo un'applicazione flessibile del codice ed un controllo della camera più trasparente, a partire da un ambiente di sviluppo software accessibile, il motore grafico Unity, rendendola fruibile attraverso un dispositivo indossabile per la realtà aumentata, nello specifico il visore Hololens.

La soluzione risultante si basa su un algoritmo sviluppato in ambito di ricerca di ottimizzazione della posa e tracking per camera RGB monoculare, eseguita su computer, ed interagisce dinamicamente con un flusso di immagini catturate dal dispositivo olografico, che oltre a costituire la fonte dell'input esegue la rappresentazione grafica di un modello identico all'oggetto che si desidera tracciare (idealmente prodotto tramite stampa tridimensionale) in modo tale da farne combaciare la posizione nello spazio percepito, cui risulterà sovrapposto grazie alla lente prismatica trasparente.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

Questa tesi di laurea prende avvio dalla necessità di precisione e flessibilità per le applicazioni di tracciamento di oggetti tridimensionali nello spazio e l'identificazione della posa in sei gradi di libertà; decenni di ricerca hanno dimostrato che è richiesto controllo minuto delle variabili in gioco, legate sia all'attrezzatura in uso che agli algoritmi, senza trascurare i dettagli degli oggetti tracciati; le librerie software disponibili in questo campo raggiungono risultati soddisfacenti in una frazione limitata dei casi d'uso, risultando inoltre opache all'analisi del codice sottostante su cui si basano, e generalmente gestiscono l'hardware di input senza permetterne usi concomitanti. Da qui nasce il desiderio di un'alternativa che, sviluppata in ambiente aperto, superi le criticità dei software disponibili senza sacrificare significativamente le prestazioni.
Nel seguito sono trattati dapprima considerazioni preliminari (cap. 1), successivamente il contesto in cui si colloca questa tesi di laurea rispetto all'ambito scientifico (cap. 2), dopodiché sono esaminati i requisiti ed i dettagli funzionali della soluzione proposta mettendo in risalto le specifiche dell'algoritmo di partenza (cap. 3); infine si trova una disamina delle caratteristiche strutturali del codice (cap.4), la discussione dei risultati ottenuti (cap. 5) e le conclusioni che si possono trarre con idee per espandere l'applicazione in futuro (cap. 6).
Come già anticipato il capitolo presente costituisce l'introduzione e si compone di sezioni che analizzano, nell'ordine, il problema pratico del tracking (sez. 1.1), la motivazione che muove questo studio (sez. 1.2) ed i limiti delle soluzioni correntemente usate dai programmatori (sez. 1.3), riassumendo infine il punto di partenza dello sviluppo con un quesito di ricerca in più punti (sez. 1.4)

1.1 Tracking in sei gradi di libertà

L'individuazione o stima della posa di un oggetto all'interno dell'immagine è un problema classico della computer vision; si distingue dall'altro problema fondamentale noto come classificazione, tuttavia il confine può non risultare sempre netto. Ad un primo sguardo entrambi i concetti partono dall'idea di individuare prima di tutto un oggetto in un'immagine, prerogativa della segmentazione tra primo piano e sfondo, si può tuttavia affermare che la stima della posa si distingue per due assunti fondamentali, il primo dei quali è il fatto che l'oggetto debba essere conosciuto con precisione; non è possibile infatti estendere la funzionalità ad una categoria, serve un singolo oggetto, limitando le capacità di classificazione. L'altro elemento

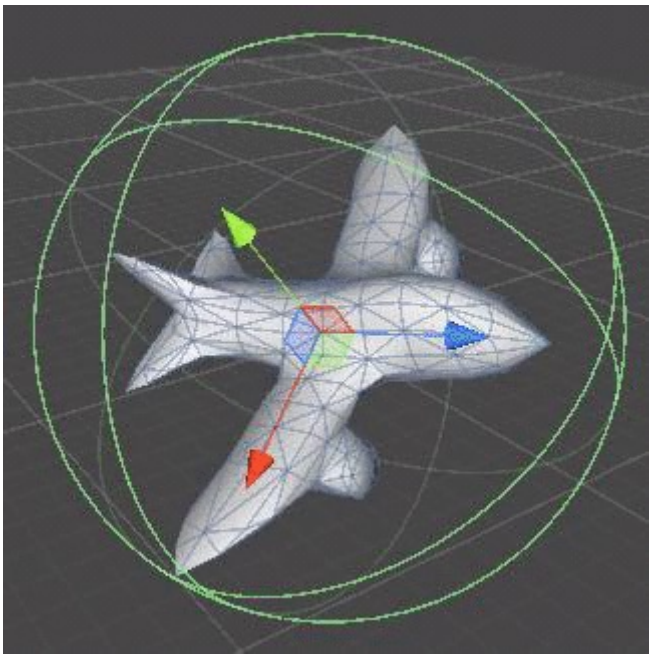

Figura 1.1: Un modello con assi e quaternioni in vista

distintivo è l'impossibilità di prescindere da una componente tridimensionale, corrispettivo digitale dell'oggetto inquadrato, modellato con precisione, se si eccettua il sottoinsieme di lavori basati su marker planari, che presentano un caso d'uso molto limitato. La formulazione classica del problema summenzionato si basa sulla possibilità di regolare la posa del modello nelle dimensioni dello spazio tridimensionale (la traslazione in tre assi perpendicolari e rotazione intorno a tre assi perpendicolari definiscono sei gradi di libertà, noti anche con l'abbreviazione dall'inglese 6DOF) perché questo assuma una posizione che approssimi con un elevato grado di precisione quella dell'oggetto rappresentato nell'immagine relativa al sensore che lo rileva; un sottoinsieme rilevante della sua applicazione a sequenze di fotogrammi è il tracking, che generalmente non corrisponde all'esatto calcolo ripetuto ogni fotogramma ma ad un tentativo di raffinamento progressivo della posa di un oggetto che varia dinamicamente, pur mantenendosi simile a se stessa per la maggior parte del tempo.

L'immagine ed il sensore sono anch'essi elementi distintivi della ricerca in quest'ambito, prevedendo l'uso di scansioni da camera monoculare, sia nello spazio di colori RGB che nel solo valore di profondità D, o fusione di sensori ottici, come nel più consueto caso del complesso RGB-D; esistono diverse soluzioni che sfruttano in concomitanza sensori basati su principi differenti come giroscopi o accelerometri (oggi presenti in ogni telefono cellulare).

Le prestazioni nel calcolo della posa, influenzate sia dall'efficienza algoritmica che dalle capacità hardware, determinano la velocità di aggiornamento di una determinata soluzione, che nei casi più desiderabili opera ad una frequenza di circa trenta fotogrammi al secondo, qualificandosi chiaramente come "real-time", eseguibile in tempo reale, ma il risultato non è sempre raggiungibile, specialmente su dispositivi portatili dotati di processori meno potenti e limitati nell'operatività anche dalla capacità della batteria, nella maggior parte dei casi d'uso, motivo per cui risulta spesso preferibile l'esecuzione dei calcoli su computer seguendo un modello centralizzato. Oggi le librerie hardware più usate nella ricerca legata alla computer vision sono

disponibili per la maggior parte dei dispositivi portatili, permettendo di non partire dallo sviluppo di software cosiddetto a basso livello, con un controllo puntuale delle componenti hardware ed i rispettivi driver di sistema, ma un tempo anche questo elemento poteva costituire una criticità.
Gli algoritmi moderni partono dal progetto modulare di una pipeline concettuale suddivisa in più passi da seguire, di cui i due cardini sono le fasi di segmentazione ed ottimizzazione o raffinamento della posa; la prima ha lo scopo di suddividere lo spazio bidimensionale di un'immagine in primo piano, ovvero l'oggetto su cui eseguire i calcoli sulla posa propriamente detta, e sfondo, mentre la fase successiva è costituita in termini pratici dalla minimizzazione di una funzione di peso derivata dalla teoria o ricavata empiricamente, che assume forme più o meno canoniche come ad esempio il problema non lineare dei minimi quadrati.

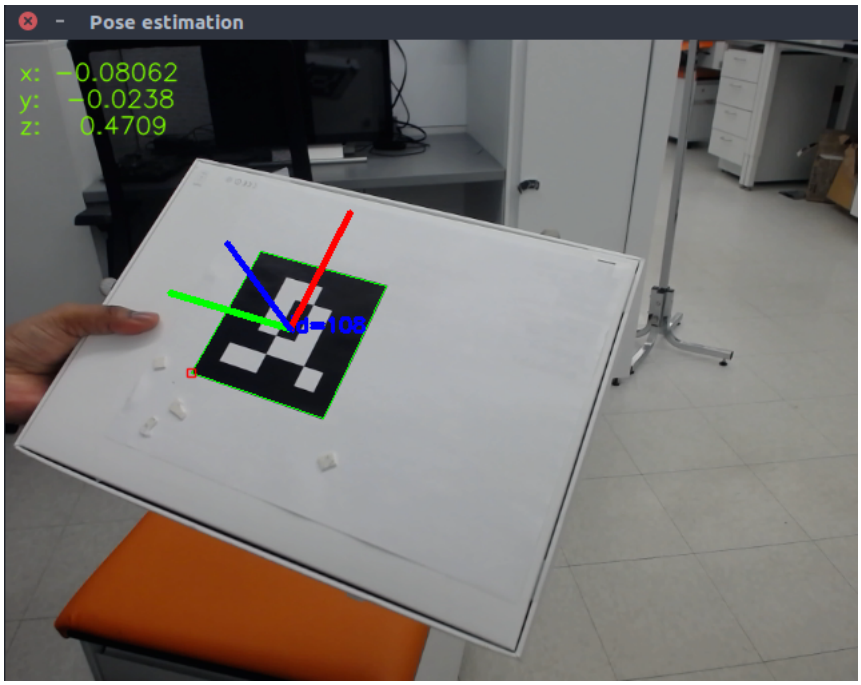

Figura 1.2: Stima della posa tramite marker

La forma che assume la segmentazione nella maggior parte dei casi trae vantaggio dal cosiddetto paradigma di machine learning, in cui l'esecuzione operativa è preceduta da una fase di allenamento o training, una funzione adattiva del software moderno che ha reso realistica una precisione prima impensabile per il tracking senza l'ausilio di marker, un tempo predominanti; questi strumenti sono oggi considerati un elemento ingombrante ed indesiderabile per la maggior parte dei casi d'uso pratici perché riducono l'applicabilità nel mondo reale dei risultati ottenuti.
Oggigiorno gli algoritmi moderni fanno leva sugli elementi distintivi tra i pixel dell'immagine; comunemente questi prendono la forma di caratteristiche locali come vertici o spigoli ed il corrispondente gradiente di colore, le cosiddette feature di basso livello, oppure elementi statistici di più alto livello, cosiddetti region-based, basate ad esempio su distribuzioni bayesiane di informazioni di luminanza o crominanza, che si usano per astrarre costrutti geometrici come punti, linee e superfici.
L'ottimizzazione della posa, che come detto ruota intorno ad una funzione di peso,

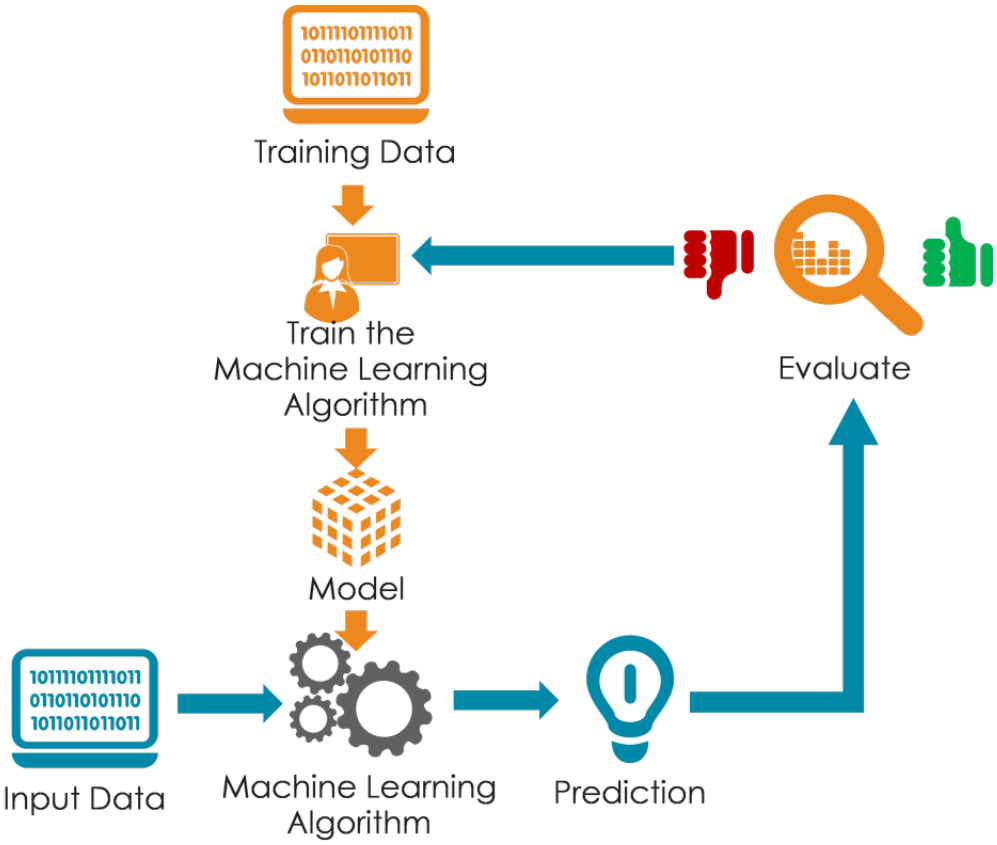

Figura 1.3: Schema del training

richiede che questa converga idealmente a zero ma soprattutto ad un valore sufficientemente vicino a zero in un numero relativamente basso di iterazioni tramite metodi numerici appropriati, selezionati ed ottimizzati in base all'hardware dedicato ma anche a caratteristiche intrinseche dell'immagine segmentata risultante dalla fase precedente di elaborazione.

1.2 Finalità ultima: tracking immersivo

La realtà virtuale immersiva è definita come un'interfaccia per una realtà simulata da computer, che delinea uno spazio tridimensionale che stimoli i sensi dell'utente in un'esperienza realistica o quanto meno coinvolgente e credibile, ricevendo al contempo meno stimoli esterni possibili per massimizzare il livello di immedesimazione soggettiva limitando al contempo il rumore costituito dalla percezione del mondo esteriore che si qualifica come distrazione. La realtà aumentata immersiva, pur mantenendo svariate somiglianze pratiche, ribalta questo paradigma integrando gli oggetti virtuali nello spazio reale; conseguentemente le esperienze più immersive devono essere svincolate dalle classiche postazioni dedicate ai regolari computer, perfino dai monitor di dispositivi portatili come gli smartphone, ed affidarsi al contrario ad elementi che agiscano sulla visione senza essere percepiti chiaramente come uno schermo, tipicamente dispositivi indossabili e più nello specifico visori o Heads-Up Display (HUD) con tecnologia see-through, sia essa basata sulla proiezio-

ne della realtà acquisita da una o più camere (Apple Vision) o al contrario rendering su elementi più o meno trasparenti come nel caso di ologrammi all'interno di lenti prismatiche (Hololens). Queste tecnologie, data la propria natura, permettono un'esperienza immersiva ed integrata in tempo reale con il mondo circostante, avvalendosi per esempio di algoritmi di tracking.

Figura 1.4: Hololens

L'evoluzione tecnologica del settore informatico di consumo ha reso disponibili dispositivi che, per quanto costosi ed ancora posseduti da un un numero piuttosto ristretto di persone, sono prodotti in massa e supportati sempre più da aziende leader nel settore, il che ha permesso la disponbilità di librerie ufficiali ad alto livello tramite le quali interagire col sistema operativo e sviluppare applicazioni in linea con gli standard dell'industria, inoltre al giorno d'oggi un dispositivo portatile è dotato di capacità di calcolo adeguate ad un gran numero di applicazioni informatiche ed in casi limite si possono compensare le carenze con il collegamento in rete. Tale concomitanza di fattori determina l'interesse della comunità imprenditoriale, per cui le nuove tecnologie presentano un'opportunità di crescita, così come del settore accademico, interessato a studiare e far avanzare lo stato dell'arte in ogni ambito tecnologico.
La barriera economica di ingresso risulta tuttavia molto elevata per i visori dedicati alla realtà aumentata, oltretutto di diffusione ancora limitata e conseguentemente votati a casi d'uso più ristretti. La disponibilità di software open-source di tracking appositamente progettato è dunque al momento carente, rendendo necessario l'impiego di librerie software commerciali come uniche soluzioni che non necessitano di riconversione.
Questo lavoro si pone l'obiettivo di esplorare le possibilità per la realizzazione di

algoritmi di tracking 6DOF avvalendosi di un visore Hololens ed opzionalmente un computer di potenza media come supporto, così come di algoritmi software derivante da applicazioni scientifiche, perciò analizzabile, documentato e modificabile secondo necessità, ma non necessariamente disponibile come pacchetto di installazione pre-compilato.

1.3 Librerie disponibili

link: https://visionlib.com, https://www.ptc.com/en/products/vuforia

Sono disponibili diverse librerie per il tracking di oggetti in sei gradi di libertà, ma le applicazioni dirette all'ambiente Hololens sono limitate a causa di incompatibilità derivanti dall'ambiente di programmazione ed esecuzione; la comunità si è mossa per creare soluzioni software, ma il campo d'azione è generalmente limitato e non sono numerose le applicazioni più raffinate come quelle dedicate alla comunità scientifica. È disponibile un numero limitato di API commerciali, che si rivelano potenzialmente limitanti da diversi punti di vista: dalle capacità parziali o assenti perché ottenute dietro pagamento del servizio (generalmente piani di sottoscrizione pensati per applicazioni aziendali di training, comunicazione e marketing) a standard operativi non riconosciuti dalla comunità tecnico-scientifica, senza poi dimenticare il segreto commerciale che impedisce l'analisi ed eventuale ottimizzazione o adattamento a bisogni specifici delle componenti centrali.
Due esempi molto noti di software dedicato alla funzione in esame sono Vuforia e Visionlib, collocati dietro "paywall", se non del tutto almeno in parte, e destinati generalmente a scopi industriali o, nel caso di Vuforia soprattutto, anche intrattenimento e marketing. Costruiti su specifiche chiuse, i summenzionati software non possono essere analizzati con lo stesso livello di dettaglio di quelli inclusi nel capitolo successivo.

Figura 1.5: Vuforia

Risulta tuttavia possibile valutare funzionalmente le prestazioni delle due librerie ed i risultati sono misti: funzionano in modo convincente in una serie limitata di circostanze e la loro efficacia può essere compromessa da diversi fattori, come riflessi sulle superfici, illuminazione ambientale, distanza, occlusione, cluttering, rumore

Figura 1.6: VisionLib

in forma di oggetti simili all'elemento soggetto a tracking. Per costruzione queste librerie operano oltretutto secondo parametri non ideali, essendo il codice alla loro base coperto da segreto ed essendo programmate per fare uso della camera in forma esclusiva durante il loro impiego.

1.4 Quesito di ricerca

Una volta ristretto il campo del tracking in generale verso una sua declinazione più immersiva, monoculare in tempo reale, è tempo di stabilire come mettere in pratica questa capacità.
Risulta quindi chiaro che il campo della computer vision in visori indossabili presenta ancora elementi di immaturità e con esse opportunità di crescita. Siamo oggi in grado di creare applicazioni di tracking in tempo reale ma questo richiede compromessi non indifferenti. Per stabilire una direzione chiara che questo lavoro possa seguire è utile esporre una research question in più punti:
- è possibile replicare lo stato dell'arte scientifico, senza compromettere il concetto originario, avvalendosi di un visore Hololens?
- la performance raggiunta sarà superiore alle soluzioni disponibili nel mondo della realtà aumentata per precisione ed accuratezza?
- l'aggiornamento della posa si potrà qualificare come real-time?
- è possibile superare i punti deboli nel processamento dell'immagine illustrati nell'ambito del software commerciale?
Per introdurre il percorso verso una risposta empirica, seguono cenni rispetto alla letteratura, con un'individuazione dei parametri rilevanti (sez.2.1), i lavori prodotti di interesse (sez. 2.2) nonché le caratteristiche distintive della base teorico/pratica di questo lavoro (cap. 2.3).

Capitolo 2

Stato dell'arte

Il tracking in sei gradi di libertà appare comunemente in letteratura scientifica ormai da decenni e le soluzioni proposte si evolvono nella teoria sottostante e con il progresso delle tecnologie in uso. Si possono osservare formulazioni diverse del problema di base che si prestano ad algoritmi matematicamente distinti, architetture software in costante mutamento che si propongono di risolverle, ottimizzazioni dell'uso di hardware disponibili nonché approssimazioni delle formule usate nei calcoli matematici in grado di massimizzare le prestazioni, al servizio di finalità sia ludiche e culturali che industriali e commerciali.
Il panorama della ricerca orientata alla computer vision si rivela più complesso e stratificato con l'avanzare del tempo e della disponibilità di hardware e software in grado di svolgere compiti come l'elaborazione bidimensionale e tridimensionale. La fase di analisi della letteratura da cui ha preso avvio questo lavoro ha richiesto un'attenta selezione di pubblicazioni sottoposte a revisione paritaria; la mole disponibile è cresciuta esponenzialmente nel corso del decennio 2010-2020 come documentato ad esempio da revisioni sistematiche della letteratura scientifica; due, per quanto non totalmente esaustive soprattutto nell'ambito degli anni più recenti, sono state particolarmente utili [25] e [37].

Nel descrivere il panorama odierno del tracking, sempre più distante dall'uso dei marker (non rilevante per questo studio) per estendere l'applicabilità al mondo reale, è possibile suddividere da un lato le soluzioni basate su deep learning e reti neurali, oggi di grande interesse ma che ancora non ha mosso passi decisivi al di fuori degli ambienti professionali più avanzati e del mondo scientifico-accademico, mentre dall'altro lato troviamo implementazioni di architetture di calcolo più classiche; una seconda suddivisione si deve fare tra gli algoritmi monoculari, operanti nello spazio dei colori standard ed esclusivamente nelle due dimensioni in cui è proiettata un'immagine, e quelli che si avvalgono di sensori di profondità (esclusivamente o soprattutto in affiancamento ad una camera tradizionale, per la cosiddetta fusione RGB-D), in grado di rappresentare con più ricchezza lo spazio tridimensionale, per loro natura più precisi ma deboli rispetto a condizioni di illuminazione, distanze superiori a 10 metri e largamente disponibili ma non per ogni caso d'uso. Infine la trattazione precisa e puntuale del soggetto non può tralasciare la distinzione dei diversi modelli teorici intorno al raffinamento della posa di un oggetto, principalmente riguardo le dovute assunzioni per semplificare il problema ed i calcoli, di cui la più specifica riguarda la posizione relativa dell'oggetto rispetto alla camera e la più generica riguarda la continuità temporale; in particolare il tracking si propone

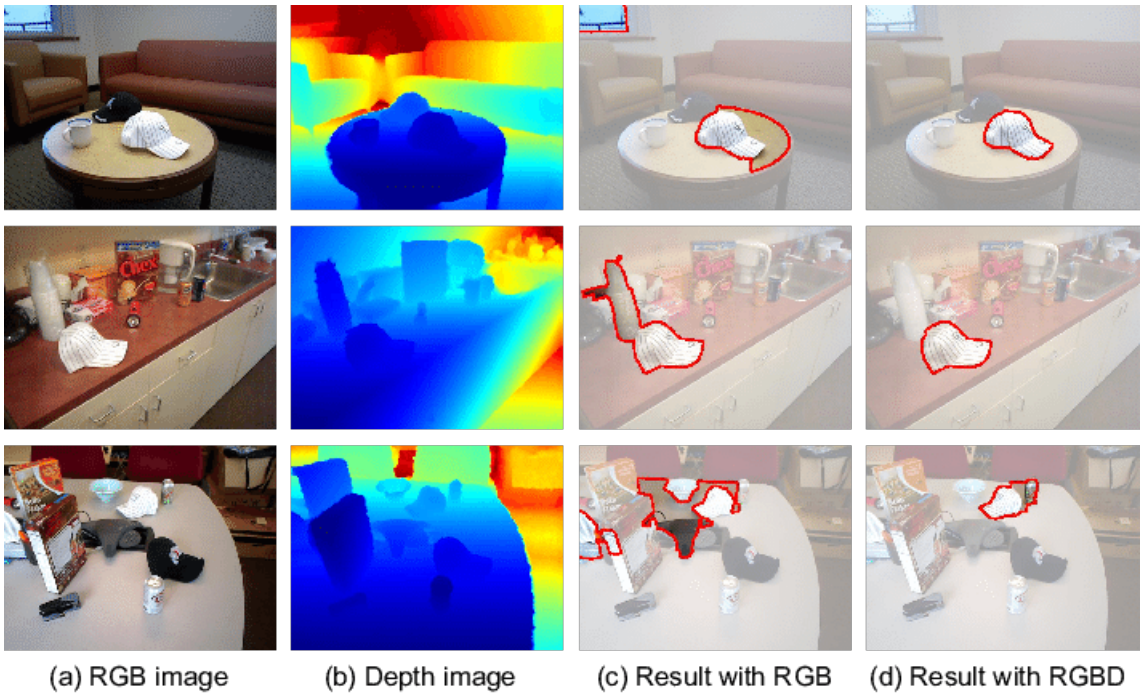

Figura 2.1: Differenza tra segmentazione di immagini RGB, RGB-D

perciò di seguire il movimento di un oggetto all'interno di un flusso di immagini partendo da una posa, stimata a partire dal fotogramma precedente a quello in esame o approssimata a priori, richiede una fase di recupero del tracciamento, spesso separata dal nucleo dell'algoritmo, in caso di interruzione della suddetta continuità. Si può constatare come i software di stima della posa e tracking appartengano a diverse macrofamiglie, la cui evoluzione si è svolta lungo svariate direttrici; sono di interesse per questo lavoro le soluzioni che hanno raggiunto un'affidabilità di livello definibile come stato dell'arte al momento della pubblicazione. Risulta altresì necessario operare una selezione all'interno della mole di soluzioni esaminate, esplicata all'inizio del terzo capitolo. Segue una disamina del progresso storico recente delle pubblicazioni inerenti il tracking.

2.1 Lavori correlati

Lo sviluppo graduale di metodi sempre più accurati di tracking si è svolto nei decenni recenti attraverso varie evoluzioni, teoriche, di hardware e software. Le prerogative alla base delle soluzioni ad oggi disponibili sono mutate spesso, a partire dalla transizione dall'uso di marker agli approcci marker-less; tra questi ultimi molta attenzione è stata riservata alla fase cruciale di segmentazione, dapprima basate sulle caratteristiche di basso livello, o feature, da individuare in un'immagine, in seguito con l'emergere di metodi statistici region-based e template matching (che prevede la preparazione di strutture di dati, i cosiddetti template, dal modello 3D) è giunta una maggiore flessibilità ed all'analisi locale si sono affiancati anche paradigmi diversi legati a strutture probabilistiche, artefatti della memorizzazione di un'immagine RGB, come gli istogrammi, declinati in formulazioni probabilistiche per segmenta-

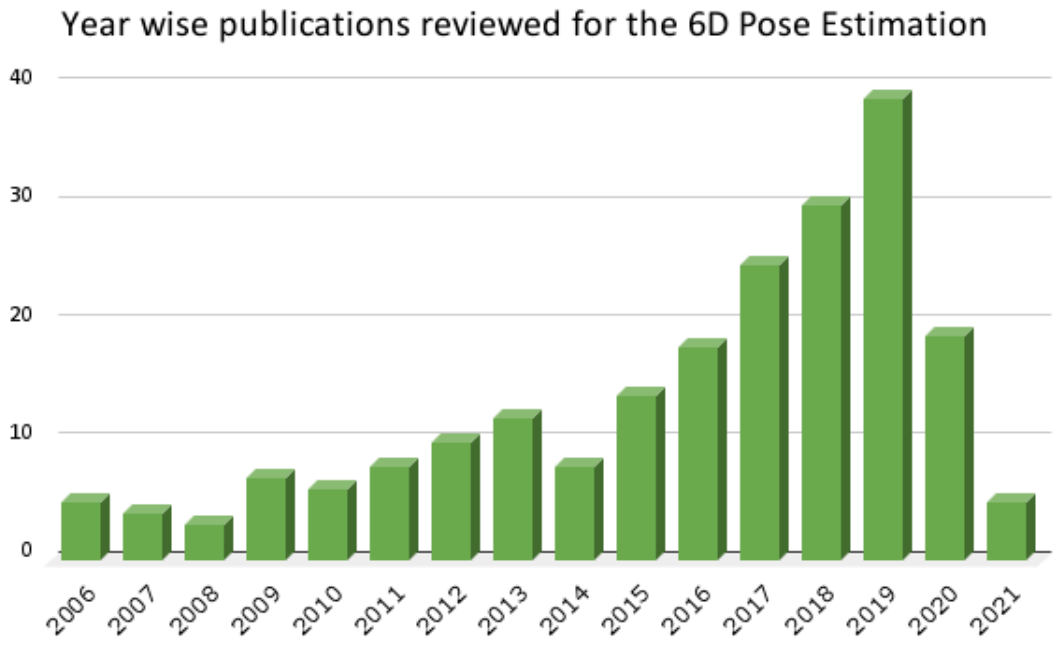

Figura 2.2: Numero di pubblicazioni legate alla stima della posa fino ai primi mesi del 2021

re le regioni di un'immagine tramite la forma, alberi di decisione e gradienti meno strettamente correlati ai pixel locali. Ognuna di queste soluzioni prevede diversi punti di forza rispetto alle classiche difficoltà nell'elaborazione dell'immagine, principalmente cluttering, occlusione, variabilità delle fonti di luce, superfici riflettenti, distorsione dovuta al movimento o motion blur.

Il caso di feature più semplice, e di concezione più datata, è costituito da quelle basate su gradienti di intensità, in grado di individuare bordi ed angoli e da questi trovare una corrispondenza di posizione ed orientamento con modelli 3D, un procedimento corrispondente all'uso di marker ma basato puramente sulle informazioni intrinseche all'oggetto esaminato; il vantaggio è la bassa latenza, il che permise il riconoscimento della posa su piccole immagini a partire dal finire degli anni 2000[3], in alcuni casi in tempo reale [23]; le feature di basso livello sono in seguito state usate principalmente all'interno di algoritmi più sofisticati, soprattutto basati su deep learning. Non molto tempo dopo è emersa la pubblicazione che ha reso stato dell'arte un approccio region-based [30], migliorava le prestazioni nel caso di texture debolmente distinguibili o assenti, eseguendo una segmentazione globale secondo il metodo dell'insieme di livello, o level-set, basandosi su presupposti teorici statistici ed agendo sui posteriori di probabilità per ogni pixel piuttosto che le probabilità di appartenenza ad una regione spaziale, tuttavia la performance in tempo reale non è stata raggiunta. Contemporaneamente si è distinto il lavoro di [27] per l'uso di immagini RGB-D ed un ben definito approccio di corrispondenza di template per una nuvola di punti identificata grazie alla componente di profondità, in tempo quasi reale, oltre che di un dataset di immagini e modelli ancora oggi in uso.

Una svolta nell'ambito del machine learning è arrivato a metà degli anni 2010, sia negli algoritmi di identificazione che in quelli di tracking 6DOF, con nuovi modelli di apprendimento basati su deep learning e reti neurali, tra cui si sono distinti per

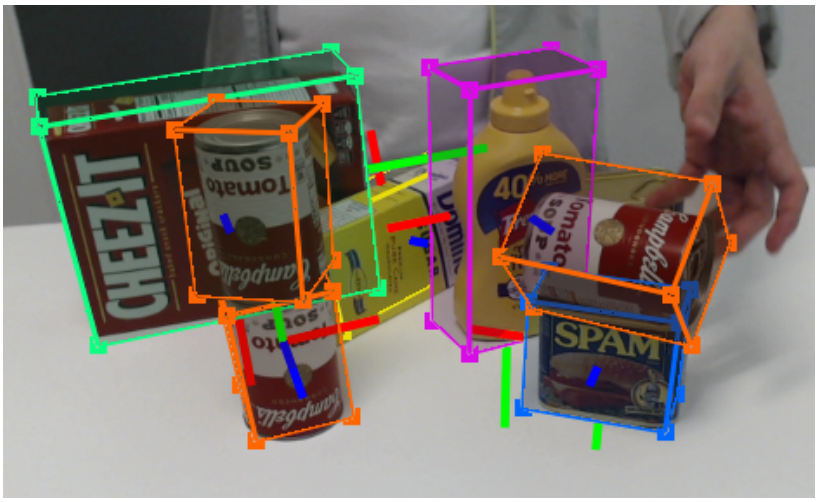

Figura 2.3: Tracking di oggetti multipli

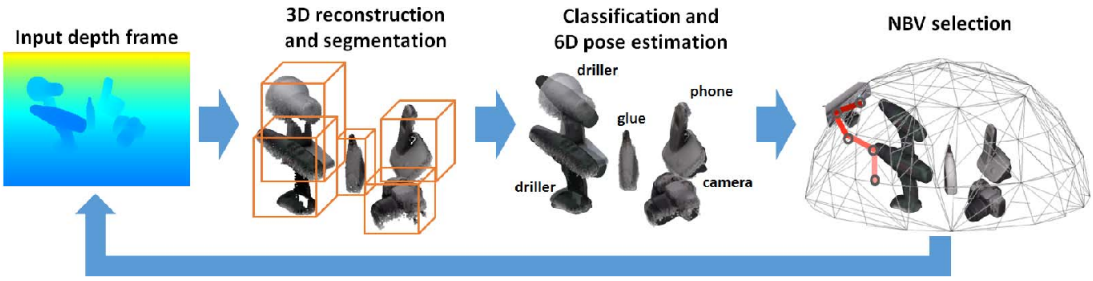

Figura 2.4: Elaborazione di template all'interno di LineMOD

la precisione lavori basati su RGB-D come [1] [18] [10], seguiti ben presto da concorrenti per immagini monoculari [5] [31]; questi lavori si allontanano dal caso d'uso in tempo reale, tranne casi sporadici [39], ma portano la precisione di rilevamento ad un livello superiore. Anche pubblicazioni basate su algoritmi più classici furono numerose, basate su funzioni di decisione più complesse e dai risultati notevolmente più precisi della generazione precedente, soprattuto nell'ambito RGB-D, declinato alla precisione [4] [29] [2] più che alla bassa latenza [12]; le soluzioni RGB di rilievo sono invece votate alla velocità di esecuzione con risultati in tempo semi-reale [13] [22]. In questo periodo inizia a delinearsi una distinzione di casi d'uso tra camere monoculari ed a fusione di colore e profondità, valorizzando la precisione di quest'ultima a scapito della latenza, ipotizzando un impiego per l'attuazione di macchinari e robot piuttosto che realtà aumentata; similmente i primi lavori basati su deep learning si sono rivelati lontani da casi pratici immediati a causa del training intensivo, temporalmente e computazionalmente, nonché basato su intervento umano per l'annotazione di una grande mole di informazioni sulla posa.

Negli anni l'accessibilità delle reti neurali è accresciuta, e con essa la proporzione di pubblicazioni al livello di stato dell'arte, ora la quasi totalità, grazie anche all'impegno rivolto allo sviluppi di metodi automatizzati per il training di algoritmi basati su deep learning, così come la riduzione dei dati numerici che richiedono intervento umano, di quasi ogni lavoro nel campo, in particolare nel campo dei dati di profondità in pubblicazioni come [21] [16] [38]. Contemporaneamente lo sviluppo procede in parallelo tra metodi RGB-D [9] [36] [11] [6] [15] e corrispettivi monoculari [20] [35] [7] [28] [33] che competono nella precisione raggiunta, facendo leva su grandi po-

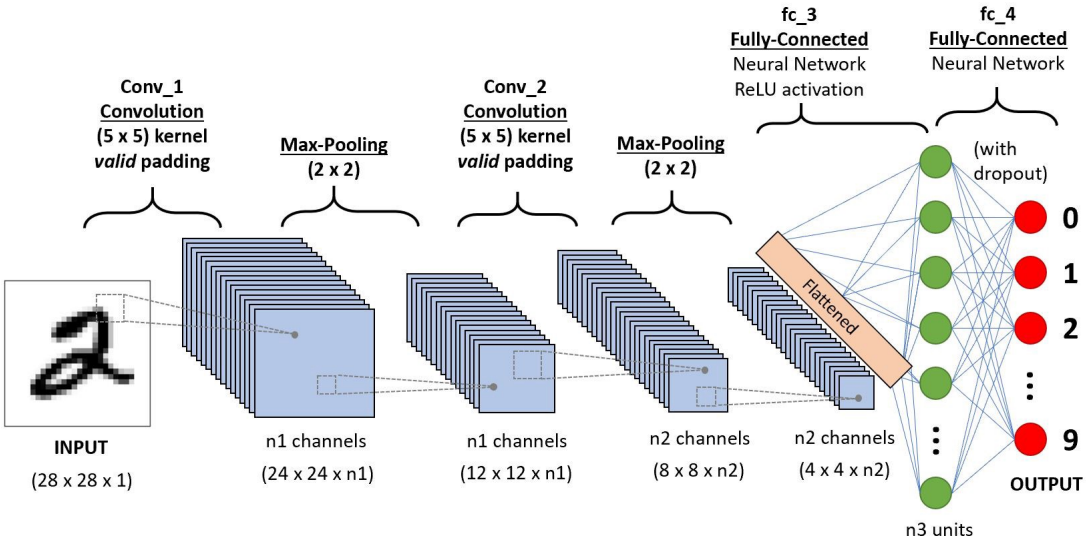

Figura 2.5: Architettura di rete neurale convoluzionale (CNN)

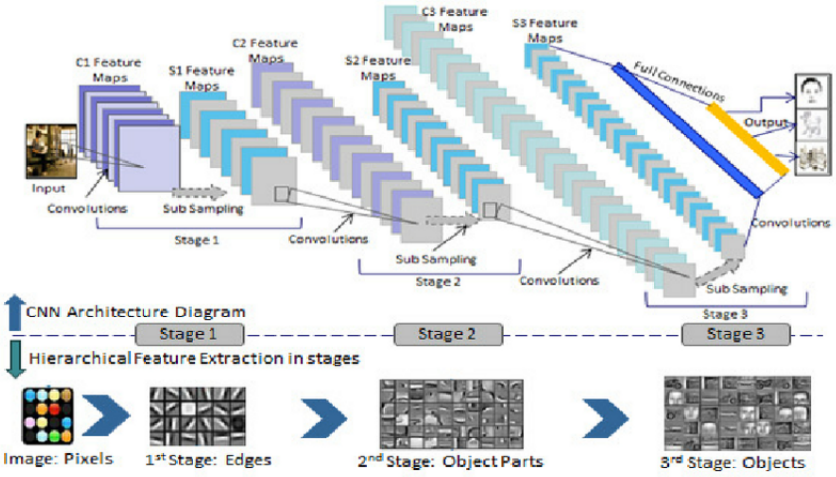

Figura 2.6: Elaborazione di feature in una rete neurale

tenze di calcolo grazie alle ultime generazioni di GPGPU, mentre una parte sempre più consistente di essi raggiunge tempi di esecuzione classificabili come real-time, dapprima con input RGB [26] [24] [8] [40] [17] [34] e più di recente anche RGB-D [14] [32]. Lo stato dell'arte nel campo di algoritmi tradizionali è stato distanziato nell'ambito della precisione di rilevamento ed il numero di pubblicazioni rilevanti si è ridotto di molto negli ultimi tempi. Si è distinto tuttavia [19], un lavoro basato su immagini monoculari e segmentazione level-set ed ottimizzazione Gauss-Newton che ha raggiunto prestazioni in tempo reale ed un grado di precisione superiore ai suoi precursori richiedendo una potenza di calcolo accessibile.

Capitolo 3

Parametri di progettazione

Scegliere un punto di partenza adeguato per lo sviluppo richiede un'analisi accurata dell'obiettivo desiderato, stabilendo una serie di requisiti il più possibile quantitativi, piuttosto che qualitativi, ed orientati agli aspetti più pratici e concreti. La necessità dominante per l'avvio del progetto è l'accessibilità, la soluzione più adeguata è quella più immediatamente integrabile nel framework desiderato, nello specifico il motore grafico Unity, il quale permette di programmare un dispositivo di tipo Universal Windows Platform (UWP) senza la necessità di lavorare su librerie di basso livello con target WinRT, disponendo oltretutto di una viewport per lo spazio tridimensionale. Se la semplificazione della programmazione embedded e grafica può essere considerata una comodità, l'utilizzo di algoritmi di computer vision già sviluppati e validati è una necessità, in quanto l'atto di tradurli dalle specifiche a codice funzionante richiede una specializzazione delle competenze ed ampio dispendio di risorse temporali. Si può ritenere quindi vantaggioso l'impiego in questo nuovo ambito di soluzioni open-source progettate originariamente per hardware classici, con lo scopo di svincolare il progresso nel settore da interessi terzi e stabilire nuovi standard più inclusivi per gli anni a venire.

Segue la necessità di contenere la richiesta di risorse di elaborazione, perché le due opzioni per l'esecuzione sono da dispositivo mobile, quindi per definizione limitato nella potenza e batteria, oppure da remoto con computer portatile, che per quanto potente dispone di un tempo limitato per l'aggiornamento, in virtù dei millisecondi

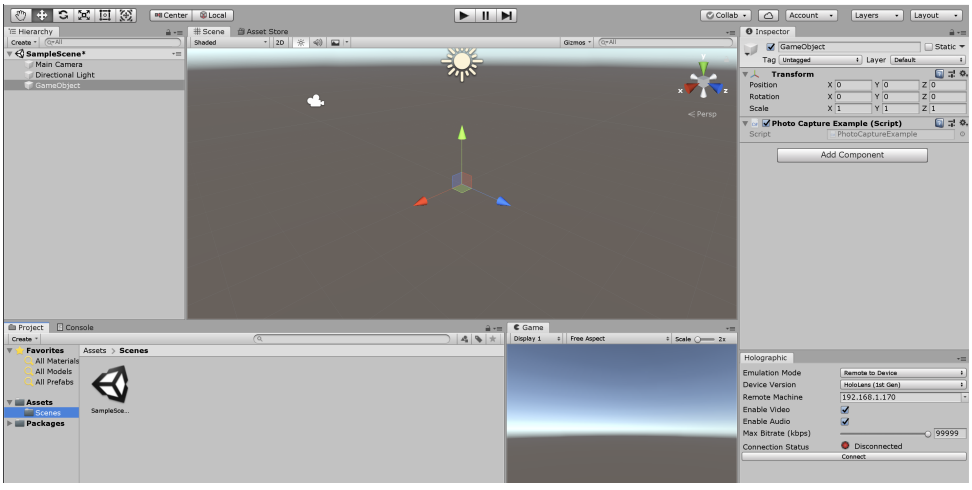

Figura 3.1: Schermata del motore grafico Unity3D

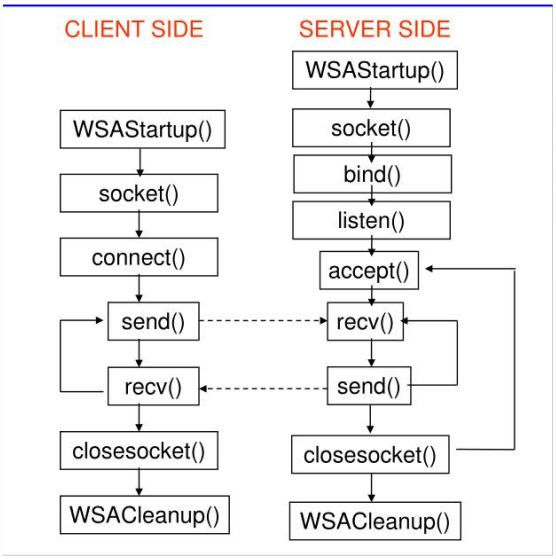

Figura 3.2: Socket TCP, modello client-server

perduti per la trasmissione su rete dell'immagine su cui eseguire i processi algoritmici (il cui risultato di risposta può essere realisticamente la matrice della posa, poche decine di byte), se si vuole mantenere l'algoritmo in tempo reale o quanto meno interattivo.
Non ultima resta la necessità di affidabilità nelle rilevazioni, sia a monte tramite calibrazione adeguata della camera, realizzata come passo preliminare all'integrazione di una soluzione di tracking, sia la precisione numerica dei risultati dell'elaborazione, legata ad aspetti di grafica, settaggi e calcoli matematici discreti. In questo contesto è importante chiarire anche che la calibrazione della camera in uso dev'essere stata eseguita a priori: lo sviluppo di procedure per ottenere le variabili intrinseche (matrice di calibrazione e coefficienti di distorsione) nonché la valutazione di algoritmi su cui dette procedure si basano cade al di fuori dello scopo di questa tesi.

3.1 Elaborazione centralizzata

Un fattore non essenziale ma che è stato considerato è la compatibilità delle librerie software da utilizzare con la piattaforma di sviluppo: Unity offre un'ampia compatibilità anche grazie all'uso di plugin ma è certamente un elemento di complessità in più.
Molti algoritmi si basano su una fase di rendering fuori schermo e, tra le risorse di memoria, di un contesto in cui effettuarlo; è stato effettuato un tentativo per riconvertire le librerie di contesto grafico alla base della soluzione poi scelta per lo studio, ma il dispendio di risorse si è rivelato infruttuoso e non resta che constatare la necessità di svolgere l'elaborazione in forma centralizzata piuttosto che come plugin.

In queste circostanze il protocollo di rete non necessita di capacità insolitamente veloci ed è risultata preferibile la programmazione di semplici socket TCP ai due

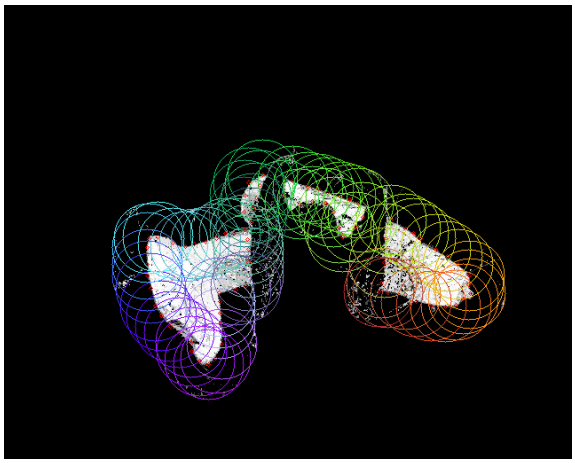

Figura 3.3: Delineazione di silhouette in RBOT

capi, programmabili in API di largo impiego sia in codice C++ (Boost::asio::tcp) che C# (System.Net.Sockets)

3.2 Classe di algoritmo in uso

Le soluzioni basate su reti neurali convoluzionali, per quanto interessanti, sono incompatibili con l'hardware in uso in quanto nella quasi totalità dei casi, specialmente escludendo quelle non adatte all'esecuzione in tempo reale, è richiesto hardware di potenza eccessiva oppure l'uso di librerie specifiche alle schede grafiche NVidia, non compatibili con la AMD a disposizione.
Hololens dispone di camera di profondità, ma è stato stabilito nelle primissime fasi di studio come preferibile prendere a riferimento il mondo del tracking monoculare, come esplicitato ed argomentato nel paragrafo successivo; similmente, la selezione di un algoritmo classico di segmentazione rispetto al deep learning ha richiesto più indagini ma è comunque risultato un parametro preliminare nello sviluppo di un'applicazione ed annessa libreria, perciò queste due direzioni guidano la selezione di una pubblicazione di riferimento adatta.
L'uso di software basato su fusione di sensori ottici si è rivelato impraticabile; è possibile accedere alla camera di profondità Hololens soltanto con codice C++ di basso livello, anziché con uno script gestito scritto con C# ed integrato pienamente in Unity. Resta teoricamente possibile integrare un plugin nativo dentro l'applicazione, perdendo però diversi vantaggi contemporaneamente: threading, driver di periferiche e funzionalità di rete, il tutto amplificato dalla piattaforma di sviluppo WinRT, con target Universal Windows platform, che crea inconsistenze nell'interazione con le componenti sviluppate in ambiente Posix/Linux, come ad esempio il rendering all'interno delle soluzioni vagliate o i socket del sistema operativo in uso.

Figura 3.4: Dataset RBOT

3.3 Conformità di RBOT

Al termine della fase di selezione, è stato deciso che il presente lavoro riadattasse una soluzione per camera monoculare proveniente da una pubblicazione datata 2018 affiancata da un repository online pubblico di codice [1]; a partire da questo si può compilare il codice sul proprio computer per installare un'applicazione dimostrativa delle capacità originali dell'algoritmo sviluppato.
La pubblicazione in esame [19], opera di Henning Tjaden et al., prende avvio da due precedenti lavori presentati consecutivamente dallo stesso autore in sede di conferenze internazionali annuali sulla computer vision: sviluppate prendendo ispirazione principalmente da PWP3D, con la segmentazione level-set e rappresentazione statistica region-based migliorandone la precisione e l'affidabilità rispetto alla presenza di cluttering. Il lavoro ora in esame li espande sotto vari livelli: innanzitutto è proposta una derivazione sistematica di un'ottimizzazione Gauss-Newton tramite la formulazione del problema in forma differente, ovvero un problema di minimi quadrati non lineare nuovamente pesato. Ciò migliora ulteriormente il tasso di convergenza e perciò la robustezza del tracking se comparato agli sviluppi evolutivi dei due anni precedenti. Secondariamente, è spiegato come la metodologia che fa uso di istogrammi appositamente concepiti possa essere estesa al tracking di oggetti multipli e le potenzialità di impiego in scenari in tempo reale affetti da rumore. Terzo punto, è stato approntato un nuovo dataset di oggetti 6DOF semi-sintetici, che copra la maggior parte delle problematiche che il software propone di risolvere.

All'atto pratico, è derivata una funzione di peso per stimare la posa 6DOF di un oggetto selezionato tra alcuni familiari (Dataset RBOT) catturati da una camera RGB semplice. L'approccio basato su regioni impiegato prevede un modello di segmentazione statistica dell'immagine derivato da molteplici regioni di immagine sovrapposte lungo il contorno di un oggetto per ogni fotogramma. Il nucleo di questo modello sono istogrammi locali di colore a consistenza temporale (acronimo dall'inglese TCLC) calcolate dalle suddette regioni di imagine, ognuno ancorato ad un'unica posizione sulla superficie dell'oggetto, il che permette di aggiornarli ad ogni fotogramma. Un'approssimazione utile per descrivere questa funzionalità è l'allineamento di due silhouette, che in realtà costituiscono rappresentazioni statistiche adeguate del modello e di una regione dell'immagine. L'obiettivo di algoritmi di questa tipologia è minimizzare la discrepanza tra le forme che rappresentano le due "silhouette" dell'oggetto e della regione che gli corrisponde in un'immagine, ricavando una maschera di segmentazione dipendente dalla posa parametrizzata per

[1] Scaricato da RBOT repository on Github

l'oggetto renderizzato sinteticamente; è richiesta l'inizializzazione delle statistiche con una posa arbitraria a partire da un fotogramma iniziale, dopodiché il modello statistico e la maschera si aggiornano progressivamente raffinando la posa stimata e convergendo verso quella osservabile nella serie di immagini.
Seppure tradizionalmente tali funzioni di peso siano state ottimizzate per mezzo di discesa di gradienti, Tjaden ha scelto di derivare uno schema di ottimizzazione di Gauss-Newton appropriato, per quanto non immediato, in quanto la funzione di peso non è nella forma tradizionale non lineare di quadrati minimi. In numerosi esperimenti, è stata dimostrata l'efficacia dell'approccio in scenari complessi e diversificati, compresi il moto sia dell'oggetto che della camera (in condizioni di motion blur minimo o assente), occlusioni parziali, grandi cambiamenti di illuminazione e confusione dello sfondo, il tutto confrontato con il precedente stato dell'arte.

Capitolo 4

Architettura del software

Il software sviluppato nell'ambito di questo progetto è dotato di una struttura modulare e può essere suddiviso in due livelli ben distinti, che divergono per visibilità all'utente, complessità e linguaggio di programmazione; il dispositivo Hololens costituisce il lato interattivo, totalmente servito da uno script in Unity, ed il computer desktop svolge il ruolo di substrato dedicato all'elaborazione; la comunicazione è infine demandata al protocollo TCP in un rapporto client-server.
L'architettura del software consta di 25 file C++, di cui 9 coppie di intestazioni e file con le funzioni corrispondenti, più 6 shader GLSL ed infine un main di esempio ampliato e modificato ai fini di questo lavoro; tra i 9 distinti moduli presenti, riportati in appendice, troviamo un wrapper di funzioni di utilità generale per matrici e 2 distintamente dedicati ad operazioni di rendering in OpenGL, più altri 6 che si suddividono i compiti legati alla posa dalla raccolta di template alle operazioni di minimizzazione delle funzioni di distanza, parallelizzate con l'uso di classi template OpenCV; successivamente sono stati aggiunti un server TCP, all'interno del file main ed infine un modulo di interfaccia incapsulato in Unity, in codice C#, demandato al client TCP, acquisizione dell'immagine e rendering del modello olgrafico. L'esecuzione di quest'ultimo può avvenire solo su sistema Windows per motivi di compatibilità con l'hardware, è stato perciò necessario operare tramite macchina virtuale per la compilazione e distribuzione.
I paragrafi seguenti saranno suddivisi secondo un criterio di priorità, perciò ne sarà dedicato più d'uno alle funzioni centrali dell'applicativo per poi chiudere l'analisi con le componenti più esposte al lato utente.

4.1 Rendering fuori schermo

La componente di RBOT che riveste il ruolo più centrale è indubbiamente quella legata al rendering off-screen, vitale al funzionamento di tutto l'applicativo e proprio per questo in relazione con gran parte degli altri moduli; tre librerie esterne costituiscono la base dell'operatività dei file rendering_engine.cpp e model.cpp (con rispettive intestazioni), ovvero OpenGL per la pipeline grafica, Assimp per la memorizzazione dei vertici e poligoni dei modelli, nonché Qt che funge da wrapper di alcuni metodi e variabili OpenGL, fornendo un contesto grafico per l'istanziamento di finestre su schermo; come accennato si è tentato di rimuovere quest'ultima per motivi di flessibilità demandando la gestione le estensioni dello standard OpenGL

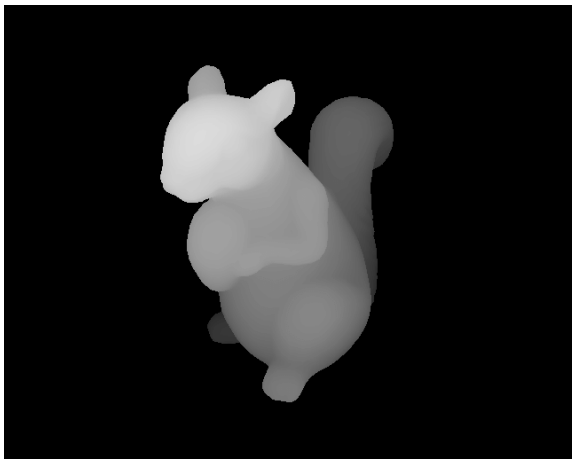

Figura 4.1: Rendering fuori schermo, mappa di profondità

tramite l'insieme di header denominato GLAD, tuttavia il contesto in uso da parte di Unity non ha comunque consentito flessibilità soddisfacente per il rendering fuori schermo e le modifiche sono state scartate.
L'oggetto model è definito secondo modalità standard per la programmazione grafica, contenendo informazioni relative a trasformata (posa), vertici con relativi indici e normali, i buffer legati ad ognuno di questi tre elementi, più dati relativi al bounding box; a livello di metodi si trovano quindi funzioni helper get e set relative a diverse di queste informazioni, una funzione di inizializzazione dei buffer ed una di normalizzazione rispetto al centro del bounding box.
L'oggetto rendering_engine astrae invece la pipeline grafica, partendo dalla definizione di un viewport e relativo contesto, posizione delle fonti di illuminazione, programmi di shading e matrici, sia la calibrazione della camera virtuale che matrici di proiezione e sguardo.I metodi presenti svolgono l'inizializzazione del contesto, render specifici per modello con texture, normali e silhouette, inoltre è possibile accedere direttamente al raster tramite downloadFrame(), proiettare la bounding box di un object su due dimensioni, e svolgere i render su più livelli di dettaglio, specificamento a benficio dei calcoli bidimensionali.
Un elemento apparentemente secondario dell'architettura ma che all'atto pratico risolve con eleganza un problema centrale all'architettura sono i vettori Qt nell'interfaccia con gli shader, perché in grado di minimizzare l'inconveniente di matrici con orientamento diverso (row-major e column-major rispettivamente) delle librerie OpenCV ed OpenGL, ruolo svolto all'interno di una classe apposita contenuta nell'header shader.hpp; ultimo è il file che memorizza le mesh.hpp che associa ad ogni mesh singola un Vertex Array Object.
L'utilizzo della pipeline di rendering è subordinata a scopi di calcolo ed infatti il caso d'uso rilevante è la renderizzazione su un framebuffer object creato appositamente seguito dalla lettura dello stesso per processare l'immagine risultante, in genere una silhouette generata con l'ausilio di una delle tre coppie di shader.
Secondariamente la componente grafica è usata per la rappresentazione su schermo della posa risultante applicata all'oggetto, a sua volta sovrimposto all'immagine in input; questo caso d'uso si è rivelato utile in fase di sviluppo ma costituisce un dispendio inutile di risorse in fase di test delle prestazioni.

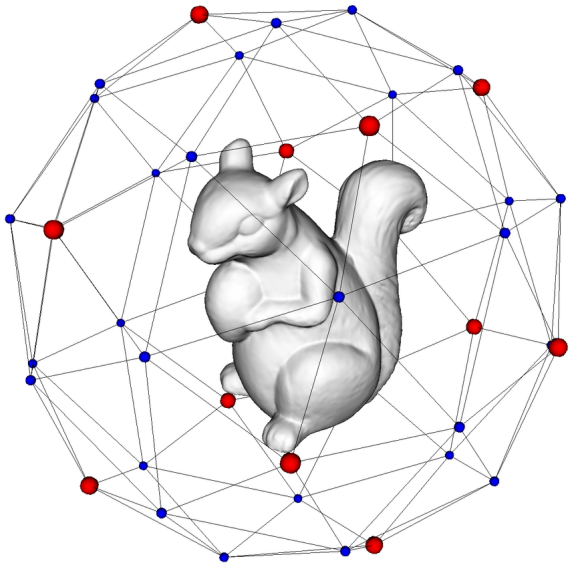

Figura 4.2: Campionamento delle prospettive per template

4.2 Strutture di dati

Una volta definito lo strato grafico di base, a questo sono sovrimposte strutture di dati utili alla successiva fase di calcolo intorno all'immagine bidimensionale per la determinazione di una trasformata appropriata da applicare al modello da renderizzare. I due oggetti astratti finalizzati al calcolo e successiva memorizzazione delle caratteristiche visive sono gli istogrammi all'interno di tclc_histograms.cpp e, per casi d'uso più limitati, la rappresentazione a template, contenuta in template_view.cpp. I template in particolare sono utili per il recupero del tracciamento una volta che questo si sia perso, mentre gli istogrammi contengono i dati fondamentali per i calcoli relativi alle regioni spaziali locali in cui applicare il processo di raffinamento della posa.Al centro di questi due, si trova il cosiddetto object, contenuto nell'omonimo file object.cpp, che racchiude al tempo stesso il modello, i template e gli istogrammi ad esso relativi, permettendone l'accesso sia alle strutture del paragrafo successivo che astraggono il processo di stima effettiva della posa.
Partendo nell'analisi dall'ultimo elemento, object, si trovano metodi per generare e recuperare i template, recuperare soltanto gli istogrammi, segnalare la perdita di tracciamento ed ottenere la soglia di qualità relativa alla posizione in output al tracking. Nel file relativo agli istogrammi, oltre all'oggetto con relativi membri e metodi per calcolo e memorizzazione propriamente detti, sono presenti delle funzioni ottimizzate per il calcolo parallelo, finalizzate all'elaborazione di istogrammi locali, unione degli stessi e calcolo del centro dell'area spaziale relativa ad un singolo istogramma. L'oggetto template_view definisce i livelli del rendering, una sorta di piramide a dettaglio crescente ed area decrescente, calcolate a partire da una funzione di Heaviside, con le relative regioni di interesse in cui applicare maschere di segmentazione e funzioni di distanza; inoltre la funzione per ottenere la Heaviside è parallelizzata fuori dall'oggetto.

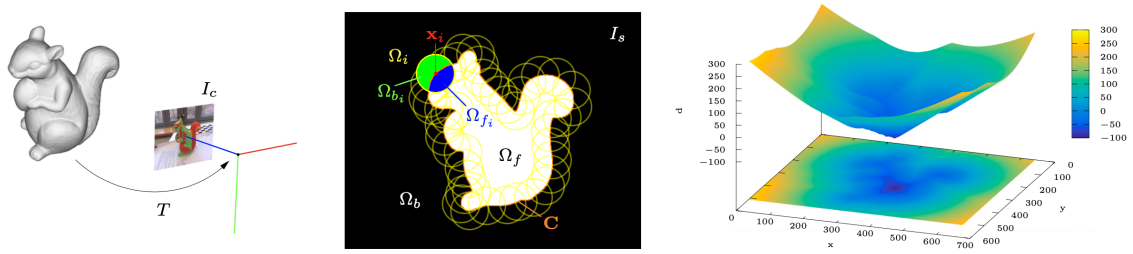

Figura 4.3: Visione ad alto livello dell'ottimizzazione region-based

4.3 Elaborazione bidimensionale, ottimizzazione e calcoli statistici

La funzione di stima della posa ha un'architettura relativamente complessa, caratterizzata dallo svolgimento di calcoli suddivisi in ulteriori sotto-moduli, tra l'object di object3d.cpp che incapsula il modello ed i relativi istogrammi e template, il file che coordina l'operatività di tutto il software tramite l'oggetto ad alto livello per la stima della posa, pose_estimator6D.cpp oltre ai già citati istogrammi in tclc_histograms.cpp ed al cosiddetto engine di ottimizzazione contenuto in optimization_engine.cpp; infine la trasformata principale per la distanza è un oggetto all'interno dell'apposito file signed_distance_transform2d.cpp.
All'interno del file pose_estimator6d.cpp è definito l'oggetto omonimo che sovrintende a tutte le funzioni legate alla stima, inizializza per primo il motore di render nonché gli object con i relativi istogrammi e template, inoltre le funzioni membro estimatePoses() e relocalize() sono chiamate rispettivamente per calcolare un'approssimazione della posa ad ogni iterazione, tramite render fuori schermo di silhouette e minimizzazione di una energy function tramite istogrammi, e reinizializzare il tracciamento una volta persa la localizzazione dell'oggetto con una ricerca esaustiva tramite template; molte definizioni ulteriori per il calcolo in parallelo sono presenti fuori dall'oggetto principale: ottimizzazioni parallele per valutazione della funzione di energia, conversione a binario di matrici, valutazione di mappe di posteriori di probabilità, template matcher, ricerca esaustiva e ricerca per vicino. Più specializzato è il ruolo di optimization_engine.cpp, con una funzione minimize() che richiama più volte runIteration() applicando due ulteriori funzioni membro per la delimitazione della ROI bidimensionale, compute2DROI(), ed il calcolo dell'ottimizzazione propriamente detta tramite applyStepGaussNewton(); l'ultimo elemento presente, fuori dall'oggetto principale e parallelizzato, è finializzato al calcolo di Jacobiani. Il file di codice rimanente, signed_distance_transform2d.com, contiene nell'oggetto omonimo la funzione di calcolo analitico numerico tramite ampio uso di derivate, più diverse ottimizzazioni parallele nelle dimensioni dello spazio bidimensionale.

4.4 TCP e Unity

Un ciclo di esecuzione è racchiuso tra l'input di un'immagine e l'output di una matrice di posa stimata; questi due passaggi richiedono la trasmissione da e verso il modulo Unity tramite un semplice socket TCP che costituisce il server. Questo è nei fatti programmato come oggetto socket nel namespace boost::asio::ip::tcp, legato ad

un context ed acceptor, impostato in ascolto al termine dell'inizializzazione del motore di rendering e le strutture di dati per la stima; una volta connesso Hololens, ogni ciclo prevede la ricezione di uno specifico numero di byte che costituiscono un'immagine 1280x720 in formato BGRA, speculare lungo l'asse x, perciò una volta converita in matrice OpenCV avviene la chiamata a pose_estimator->estimatePose(), si calcola la divisione tra la posa precedente e la corrente dopodiché il risultato è inviato all'indirizzo del dispositivo. Il ciclo si ripete fino al comando di terminazione dell'applicazione.
Lo scopo delle componenti in Unity è il coordinamento delle operazioni di acquisizione immagini e la trasmissione delle immagini al computer oltre all'effettiva rappresentazione del modello in forma olografica sovrimposta all'ambiente fisico circostante. Lo script è eseguito sul dispositivo Hololens, iniziando dall'invio dell'immagine catturata dalla fotocamera, al render del modello, non appena la pipeline per la stima della posa invia il risultato dei propri calcoli, i parametri geometrici per allineare l'ologramma all'oggetto reale.
Anche in questo caso è attribuito al codice un socket TCP (socket sincrono bloccante), cui spetta il ruolo di client, ed i dati ricevuti sono processati per divenire compatibili col differente linguaggio e la rappresentazione interna al motore grafico delle variabili geometriche, ricostruendo quindi una matrice dal flusso di byte TCP e scomponendola adeguatamente nelle corrispondenti trasformazioni rigide, che sono infine applicate al modello renderizzato nelle lenti olografiche.

Capitolo 5

Risultati e discussione

Giungendo alla fase di valutazione, si può suddividere quest'ultima in considerazioni preliminari, prevalentemente qualitative, e la disamina principale su base quantitativa, necessaria per trarre le conclusioni rispetto all'intero progetto e confrontare oggettivamente la soluzione proposta con quelle documentate in passato. Un assunto fondante del caso d'uso dell'applicazione studiata è la staticità del soggetto dotato di visore: essendo uno studio preliminare l'immersività completa non era un obiettivo fondante quanto la dimostrazione di applicabilità ed affidabilità di un algoritmo di tracking nell'ambito di un dispositivo portatile concentrando l'attenzione sull'incertezza della posizione dell'osservatore, non fissa come nel caso di un dispositivo posizionato su un treppiede.
La prima e più immediata osservazione è la limitata applicabilità della libreria di tracking in un dispositivo indossabile come Hololens: la definizione medio-alta per uno standard commerciale della fotocamera frontale, unita possibilmente ad una procedura di calibrazione a monte sistematica ma non avanzata secondo standard odierni, ed infine un singolo caso d'uso altamente dinamico in relazione al dispositivo stesso, hanno determinato un risultato carente nel comparto del tracking, che risulta in tal senso ancora lontano dalla piena operatività.
L'algorimo scelto, ancorché semplice da applicare, necessita di impostazioni accurate per l'ottimizzazione delle risorse di calcolo, all'atto pratico causando un funzionamento insoddisfacente alla presenza di immagini di grandi dimensioni, oggetti con conta poligonale troppo bassa e distanze diverse.
Un'altra considerazione è il training, che richiede alcuni minuti per essere completato, svolto appena prima dell'esecuzione; sarebbe preferibile una soluzione basata su caching, ma lo stato attuale risulta comunque superiore al training più intensivo (nell'ordine delle ore) richiesto da algoritmi più avanzati.
La precisione risente molto dei numeri in ingresso, è prevista una fase di normalizzazione della posa che può facilmente introdurre un errore sistematico nel risultato in output. La convergenza, e conseguentemente l'accuratezza, al contrario è elevata, fintanto che l'esecuzione dell'algoritmo prosegue correttamente, spesso accade che la posa rilevata sia zero in caso di uscita dai parametri operativi.
Per riassumere le tesi esposte, si fa riferimento al quesito di ricerca: il tentativo di avvicinarsi allo stato dell'arte ha rivelato difficoltà tecniche nell'applicare algoritmi e principi ad un ambiente dinamico e poco controllato rispetto ad un laboratorio, nella forma di immagini di qualità inferiore all'ideale e conseguente scarsa affidabilità dell'elaborazione nello spazio delle immagini; la precisione raggiunta nella maggior

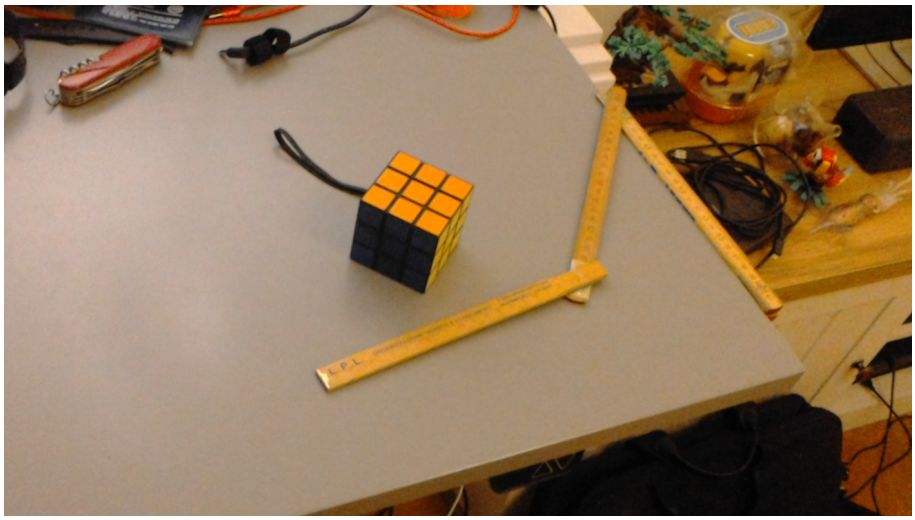

Figura 5.1: Oggetto reale, cubo

modello	errore medio (gradi)	deviazione standard	frame rate
cartone di uova	2,2	0,32	19
tetraedro	3,1	0,31	20
cubo	2,8	0,37	20

Tabella 5.1: Misurazioni corrispondenti ai modelli testati

parte dei casi è in linea con le aspettative, ma non migliora sostanzialmente le soluzioni disponibili, così come le prestazioni real-time non sono state raggiunte, anche se si possono qualificare come prossime; infine la soluzione proposta ha una flessibilità superiore al software commerciale nell'uso di hardware e software, ma non partendo dall'interesse per l'esperienza d'uso si dimostra immatura nell'applicazione efficace di impostazioni legate alle variabili di esecuzione.

Una volta esplorate le caratteristiche qualitative, è doveroso focalizzarsi sulle misure, in particolare l'attenzione è stata rivolta in due aree: il frame rate e la precisione della convergenza della posa. La misurazione del primo è semplice, ma il risultato è una media di numeri intorno ad una quantità relativamente stabile; al contrario la seconda ha richiesto un approccio più sistematico al problema.

Per iniziare, è nota la risoluzione dell'immagine, in quanto parametro selezionabile all'interno del codice. Si è scelto 1280x720, una misura standard contraddistinta da stabilizzazione dell'immagine e come si evince dalla pagina dedicata sul sito Microsoft [1] il campo visivo è di 45 gradi. Insieme alle misure in centimetri degli oggetti sottoposti a tracking vi è una corrispondenza tra gradi e pixel a specifiche distanze. In virtù del principio di studiare casi d'uso il più possibile vicini ad applicazioni reali, la tecnica di misurazione è rudimentale e meno ripetibile delle pubblicazioni sottoposte a revisione paritaria, ma può dare un'indicazione generale dei risultati raggiunti e la direzione da intraprendere per eventuali sviluppi futuri.

I modelli usati per la valutazione sono una commistione di elementi geometrici ed oggetti della vita comune: sono stati selezionati un cubo, un tetraedro dalle facce non lisce ed un cartone delle uova; due di questi provengono dal dataset ufficiale pubblicato insieme all'articolo originale sull'algoritmo RBOT chiamato per l'appun-

[1] Raggiungibile a Hololens camera official reference

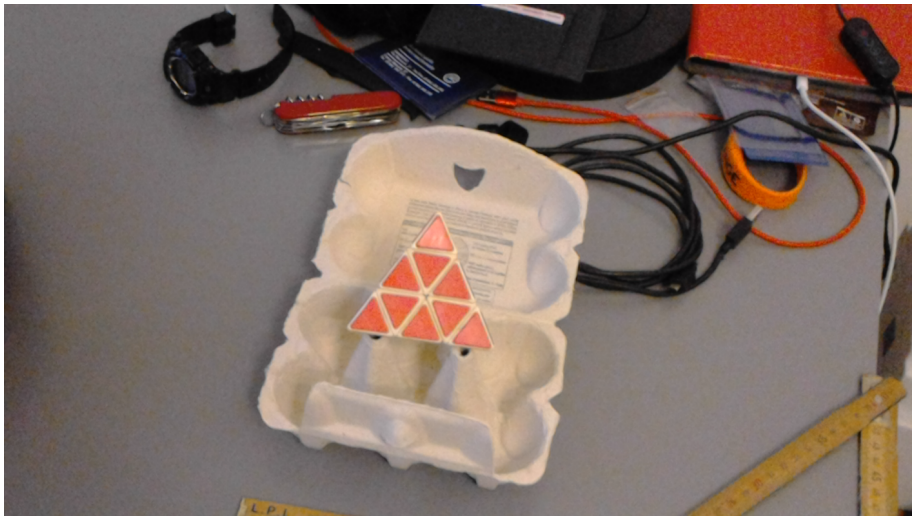

Figura 5.2: Oggetto reale, tetraedro

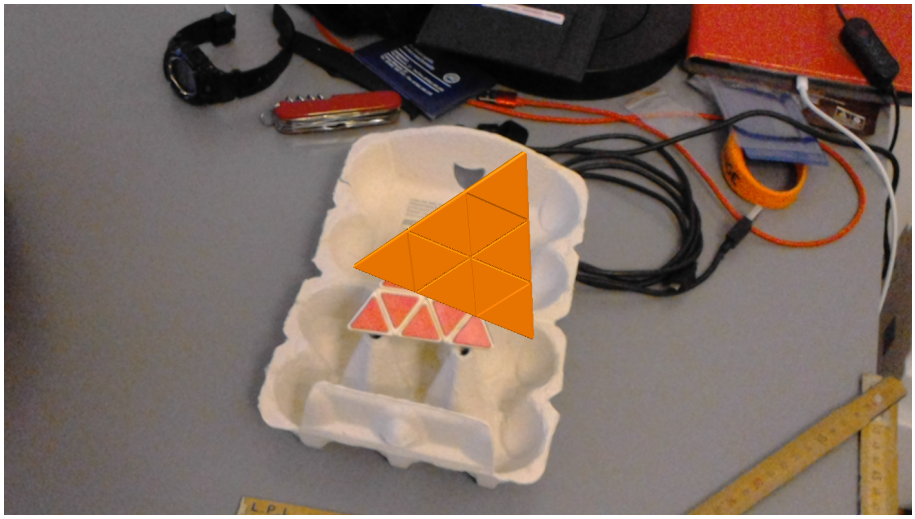

Figura 5.3: Tracking in corso, tetraedro

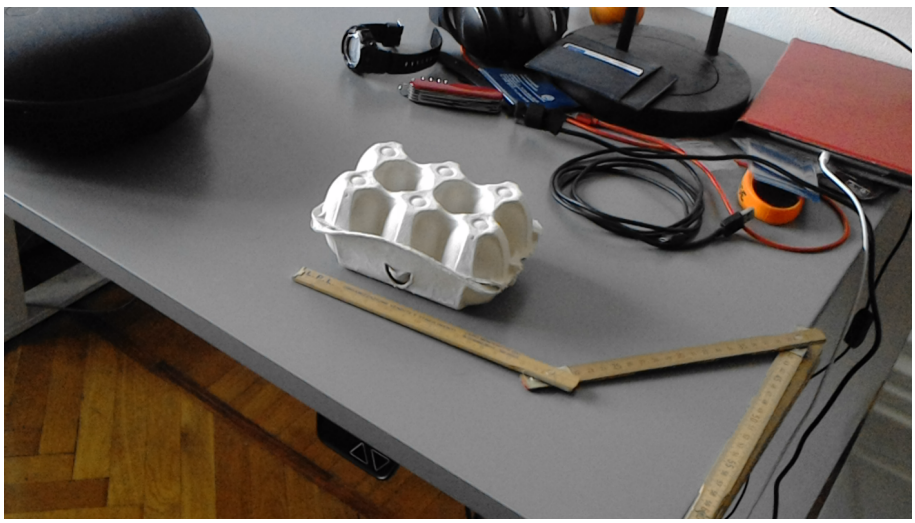

Figura 5.4: Oggetto reale, cartone di uova

Figura 5.5: Tracking in corso, cartone di uova

to RBOT-dataset.
È possibile osservare la correlazione tra il corretto funzionamento e la conta poligonale, incompatibile con l'uso di modelli semplici. La spiegazione è da ricercarsi nella rappresentazione delle informazioni statistiche sulla superficie dell'immagine, basata su template ricavati da campionamento dei vertici; questo sistema si rivela efficace in presenza di bordi frastagliati ed anche di superfici curve, ma non offre vantaggi significativi per le forme più basilari.
Un'altra considerazione riguarda le fluttuazioni nella precisione di misura, da ascrivere all'uso di oggetti reali e non render sintetici per la stima della posa, come già detto l'algoritmo è stato dimostrato resiliente al movimento, ma per misurazioni su immagini sintetiche, con sovrimposto un render alle sequenze registrate. Il motion blur si è dimostrato problematico, inficiando la precisione al punto da portare a casi di non convergenza.

Capitolo 6

Conclusione e sviluppi futuri

Le conclusioni di questo studio sono le seguenti: la metodologia di tracking con camera monoculare, per quanto ubiqua, si è rivelata carente in situazioni di alta dinamicità; è stata mostrata solidità nella ricerca scientifica in condizioni di movimento rispetto ad immagini sintetiche piuttosto che oggetti fisici, tuttavia la messa a fuoco è un supporto fondamentale a questi algoritmi e la stabilizzazione ottica deve ancora raggiungere livelli superiori.
Il ricalcolo dei template campionati tra i vertici sulla superficie di un modello sta alla base del recupero del tracking in seguito ad occlusione totale ma mostra limiti evidenti in quanto dipende fin troppo dalla vicinanza della posa di partenza rispetto a quella da stimare.
La convergenza del metodo è stata nuovamente mostrata ma al tempo stesso è stata osservata una vulnerabilità alla variazione tra frame, che porta alla perdita temporanea di tracking cui segue la necessità di reinizializzazione; come detto quest'ultima non si è rivelata pienamente affidabile limitando la durata effettiva dell'applicazione dell'algoritmo.
Inoltre la latenza non si è rivelata essere un problema in concomitanza con un'altra caratteristica del flusso di esecuzione, ovvero l'utilizzo di uno script all'interno del motore grafico Unity, che ha determinato una frequenza di acquisizione dell'immagine relativamente bassa.
Quest'applicazione ha mostrato falle ma anche potenzialità per sviluppi futuri: a partire dall'uso di dispositivi diversi ed attivamente supportati, alla comunicazione in rete ed infine al potenziamento del rilevamento di oggetti non tracciati, possibilmente da posizione arbitraria, tramite moduli più sofisticati. Sarebbe eventualmente possibile in futuro fare leva sulla disponibilità di sensore di profondità del dispositivo Hololens.

Appendice A

Codice

A.1 Tabulati di codice

A.1.1 Libreria su computer RBOT

```
#include <QApplication>
#include <QThread>
#include <opencv2/core.hpp>
#include <opencv2/highgui.hpp>
#include "boost/asio.hpp"
#include "boost/asio/io_context.hpp"
#include "boost/asio/ip/tcp.hpp"
#include <string>

#include "object3d.h"
#include "pose_estimator6d.h"

using namespace std;
using namespace cv;

cv::Mat drawResultOverlay(const vector<Object3D*>& objects,
    const cv::Mat& frame)
{
    // render the models with phong shading
    RenderingEngine::Instance()->setLevel(0);

    vector<Point3f> colors;
    colors.push_back(Point3f(1.0, 0.5, 0.0));
    //colors.push_back(Point3f(0.2, 0.3, 1.0));
    RenderingEngine::Instance()->renderShaded(vector<Model
        *>(objects.begin(),\\
    objects.end()), GL_FILL, colors, true);

    // download the rendering to the CPU
    Mat rendering = RenderingEngine::Instance()->
        downloadFrame(RenderingEngine::RGB);
```

```
    // download the depth buffer to the CPU
    Mat depth = RenderingEngine::Instance()->downloadFrame(
        RenderingEngine::DEPTH);

    // compose the rendering with the current camera image
        for demo purposes (can be done more efficiently
        directly in OpenGL)
    Mat result = frame.clone();
    for(int y = 0; y < frame.rows; y++)
    {
        for(int x = 0; x < frame.cols; x++)
        {
            Vec3b color = rendering.at<Vec3b>(y,x);
            if(depth.at<float>(y,x) != 0.0f)
            {
                result.at<Vec3b>(y,x)[0] = color[2];
                result.at<Vec3b>(y,x)[1] = color[1];
                result.at<Vec3b>(y,x)[2] = color[0];
            }
        }
    }
    return result;
}

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    // camera image size
    int width = 1280;
    int height = 720;
    //int width = 1408;
    //int height = 792;

    // near and far plane of the OpenGL view frustum
    float zNear = 10.0;
    float zFar = 10000.0;

    // camera instrinsics
    Matx33f K = Matx33f(1031.328, 0, 679.696, 0, 1034.549,
        394.479, 0, 0, 1);
    Matx14f distCoeffs =  Matx14f(0.231, -0.367, -0.0016,
        -0.0013);

    // distances for the pose detection template generation
    vector<float> distances = {200.0f, 400.0f, 600.0f};
```

```
// load 3D objects
vector<Object3D*> objects;
objects.push_back(new Object3D("data/eggbox.obj", 15,
   -0, 500, 195, -10, -20, 1.0, 0.55f, distances));
//objects.push_back(new Object3D("data/tetrahedron.obj",
    15, -35, 515, 55, -20, 205, 1.0, 0.55f, distances));
//objects.push_back(new Object3D("data/cube.obj", 15,
   -35, 515, 55, -20, 205, 1.0, 0.55f, distances));


//objects.push_back(new Object3D("data/a_second_model.
   obj", -50, 0, 600, 30, 0, 180, 1.0, 0.55f, distances2
   ));

// create the pose estimator
PoseEstimator6D* poseEstimator = new PoseEstimator6D(
   width, height, zNear, zFar, K, distCoeffs, objects);

// move the OpenGL context for offscreen rendering to
   the current thread, if run in a seperate QT worker
   thread (unnessary in this example)
//RenderingEngine::Instance()->getContext()->
   moveToThread(this);

// active the OpenGL context for the offscreen rendering
    engine during pose estimation
RenderingEngine::Instance()->makeCurrent();

int timeout = 0;

bool showHelp = true;

 // NETWORKING
 boost::system::error_code error;
 boost::asio::streambuf test_buffer;
 int port = 27014;

 boost::asio::io_context io_cont;
 boost::asio::ip::tcp::acceptor acceptor(io_cont, boost
    ::asio::ip::tcp::endpoint(boost::asio::ip::tcp::v4()
    , port));

 const string msg = "Hello From Server!\r\n\r\n";

 boost::asio::ip::tcp::socket sock(io_cont);
 cout << "Accept connection at " << port << endl;
 try {
     acceptor.accept(sock);
```

```
        cout << "Connected" << endl;

        boost::asio::read_until(sock, test_buffer, "\n");
        string in_msg = boost::asio::buffer_cast<const char
            *>(test_buffer.data());
        cout << in_msg << endl;

        boost::asio::write( sock, boost::asio::buffer(msg)
            );
    } catch (std::exception& e) {
        cerr << e.what() << endl;
        return 1;
    }

Matx44f oldPose = objects[0]->getPose(); // to normalize
Mat frame;
Mat frameDouble;
int i = 0;
int j = 0;
while(true)
{
    boost::asio::streambuf receive_buffer;

    cout << "i%3: " << i%3 << endl;
    if (!(i%7)) {
        i = 1;

         try {
            boost::asio::read(sock, receive_buffer,
               boost::asio::transfer_at_least(4460544),
               error);

            const uchar* mid = boost::asio::buffer_cast<
               const uchar*>(receive_buffer.data());
            uchar* data = const_cast<uchar*>(mid);
            Mat net_frame = Mat(1408, 792, CV_8UC4, data
               );
            flip(net_frame, net_frame, 1);

            frame = net_frame
            cout << "Input received" << endl;
        } catch (std::exception& e) {
            cerr << e.what() << endl;
            return 1;
        }

        string num = to_string(++j);
```

```
        //frame = imread("data/sequence/eggbox/
            CapturedImage" + num + ".png");
        //frameDouble = imread("data/sequence/eggbox/
            CapturedImage" + num + ".png");
        cout << "j = " << j << endl;
    } else i++;

    // the main pose update call
    poseEstimator->estimatePoses(frame, false, true);

    Matx44f deltaPose = objects[0]->getPose()*oldPose.
        inv();
    oldPose = objects[0]->getPose();
    std::vector<float> poseVec(deltaPose.val, deltaPose.
        val +16*sizeof(float));
    // send pose to device
    try {
        boost::asio::write( sock, boost::asio::buffer(
            poseVec) );
        cout << "Server sent pose to Client!" << endl;
    } catch (std::exception& e) {
        cerr << e.what() << endl;
    }
    cout << deltaPose << endl;
    //cout << objects[0]->getPose() << endl;

    // render the models with the resulting pose
        estimates ontop of the input image
    Mat result = drawResultOverlay(objects, frame);

    // save image for measures, comment for performance
    string index = to_string(++j);
    imwrite("data/sequence/result/CapturedImage" + index
        + ".png", result);

    if(showHelp)
    {
        putText(result, "Press '1' to initialize", Point
            (150, 250), FONT_HERSHEY_DUPLEX, 1.0, Scalar
            (255, 255, 255), 1);
        putText(result, "or 'c' to quit", Point(205,
            285), FONT_HERSHEY_DUPLEX, 1.0, Scalar(255,
            255, 255), 1);
    }

    imshow("result", result);

    int key = waitKey(timeout);
```

```
        // start/stop tracking the first object
        if(key == (int)'1')
        {
            poseEstimator->toggleTracking(frame, 0, false);
            poseEstimator->estimatePoses(frame, false, false
                );
            timeout = 1;
            showHelp = !showHelp;
        }
        if(key == (int)'2') // the same for a second object
        {
            //poseEstimator->toggleTracking(frame, 1, false)
                ;
            //poseEstimator->estimatePoses(frame, false,
                false);
        }
        // reset the system to the initial state
        if(key == (int)'r')
            poseEstimator->reset();
        // stop the demo
        if(key == (int)'c'  || j == 30)
            break;
    }
    // After chatting close the socket
    sock.close();

    // deactivate the offscreen rendering OpenGL context
    RenderingEngine::Instance()->doneCurrent();

    // clean up
    RenderingEngine::Instance()->destroy();

    for(int i = 0; i < objects.size(); i++)
    {
        delete objects[i];
    }
    objects.clear();

    delete poseEstimator;
}



#ifndef MODELH
#define MODELH

#include <QOpenGLBuffer>
#include <QOpenGLShaderProgram>
```

```
#include <opencv2/core.hpp>
#include <opencv2/imgproc.hpp>

#include "transformations.h"

/**
 * A 3d model class based on the ASSIMP library mostly
   implemented
 * wrt the OBJ/PLY file formats. The class provides
   functions to load the
 * model data from a specified file, drawing the model with
    OpenGL
 * as well as calculating the bounding box of the model and
    setting
 * individual vertex colors. The model data is uploaded to
   the GPU in
 * form of VertexBufferObjects.
 */
class Model
{

public:
    /**
     * Constructor loading the 3d model data from a given
       OBJ/PLY file.
     * The initial pose is computed from 3 translation
       parameters tx, ty, tz
     * and 3 Euler angles alpha, beta, gamma. Here, the
       overall rotation
     * matrix is composed as R(alpha)*R(beta)*R(gamma).
     * The bounding box is also calculated during
       initialization.
     *
     * @param objFilename  The relative path to an OBJ/PLY
       file describing the model.
     * @param tx  The models initial translation in X-
       direction relative to the camera.
     * @param ty  The models initial translation in Y-
       direction relative to the camera.
     * @param tz  The models initial translation in Z-
       direction relative to the camera.
     * @param alpha  The models initial Euler angle
       rotation about X-axis of the camera.
     * @param beta  The models initial Euler angle rotation
        about Y-axis of the camera.
     * @param gamma  The models initial Euler angle
       rotation about Z-axis of the camera.
```

```
     * @param scale  A scaling factor applied to the model
       in order change its size independent of the original
        data.
     */
    Model(const std::string modelFilename, float tx, float
       ty, float tz, float alpha, float beta, float gamma,
       float scale);

    ~Model();

    /**
     * Draws the model with a given shader programm and
     * a specified OpenGL data primitive type using VBOs.
     *
     * @param program    The shader programm to be used.
     * @param primitives The primitive type that shall be
       used for drawing (e.g. GL_POINTS, GL_LINES,...). The
        default value is set to GL_TRIANGLES.
     */
    void draw(QOpenGLShaderProgram *program, GLint
       primitives = GL_TRIANGLES);

    /**
     * The 3d data is packed into VOBs and uploaded to the
       GPU.
     * Should be called after a valid OpenGL context exists
       .
     * Must be called before a model can get rendered!
     */
    void initBuffers();

    /**
     * Must be called to start pose tracking, in order to
       indicate
     * that a set of tclc-histograms has been filled, such
       that
     * the pose of the corresponding 3D Object can be
       estimated.
     * It is also important for rendering a common scene
       mask in within
     * the rendering engine. Here only initialized models
       are drawn.
     */
    void initialize();

    /**
     * Tells whether the model has been initilaized for
       tracking.
```

```
 *
 *  @return True if it has been initialized and false
    otherwise.
 */
bool isInitialized();

/**
 *  Returns the current 6DOF rigid body transformation
 *  of the model in form of a 4x4 float matrix
 *  T_cm = [r11 r12 r13 tx]
 *         [r21 r22 r23 ty]
 *         [r31 r32 r33 tz]
 *         [  0   0   0  1],
 *  describing the transformation from model coordinates
     X_m
 *  into camera coordinates X_c.
 *
 *  @return  The current 6DOF pose of the model.
 */
cv::Matx44f getPose();

/**
 *  Sets the current model pose to a given 6DOF rigid
    body
 *  transformation in form of a 4x4 float matrix
 *  T_cm = [r11 r12 r13 tx]
 *         [r21 r22 r23 ty]
 *         [r31 r32 r33 tz]
 *         [  0   0   0  1],
 *  describing the trandformation from object
    coordinates X_m
 *  into camera coordinates X_c.
 *
 *  @param  T_cm The new 6DOF pose of the model.
 */
void setPose(const cv::Matx44f &T_cm);

/**
 *  Sets a new initial model pose to a given 6DOF rigid
    body
 *  transformation in form of a 4x4 float matrix
 *  T_cm = [r11 r12 r13 tx]
 *         [r21 r22 r23 ty]
 *         [r31 r32 r33 tz]
 *         [  0   0   0  1],
 *  describing the trandformation from object
    coordinates X_m
 *  into camera coordinates X_c.This pose is applied
```

```
    when reset()
 *  is called.
 *
 *  @param  T_cm The new initial 6DOF pose of the model.
 */
void setInitialPose(const cv::Matx44f &T_cm);

/**
 *  Returns the normalization matrix of the model. In
    the current
 *  implementation this matrix translates the model such
     that the
 *  center of its 3D bounding box is its origin and
    applies the
 *  prescibed scaling factor.
 *
 *  @return The normalization matrix of the model,
    applied before its 6DOF pose.
 */
cv::Matx44f getNormalization();


/**
 *  Returns the left (min(X0,... Xn-1)) bottom (min(Y0
    ,... Yn-1))
 *  near (min(Z0,... Zn-1)) corner of the unnormalized
    bounding box
 *  of the model.
 *
 *  @return  The left bottom near corner of the bounding
     box of the model.
 */
cv::Vec3f getLBN();

/**
 *  Returns the right (max(X0,... Xn-1)) top (max(Y0,...
     Yn-1))
 *  far (max(Z0,... Zn-1)) corner of the unnormalized
    bounding box
 *  of the model.
 *
 *  @return  The right top far corner of the bounding
    box of the model.
 */
cv::Vec3f getRTF();

/**
 *  Returns the scaling factor specified in the
```

```
    contructor.
 *
 *  @return  The prescibed scaling factor.
 */
float getScaling();

/**
 *  Returns a vector containing all unnormalized 3D
    model
 *  verticies [X_m, Y_m, Z_m].
 *
 *  @return  A vector containing all unnormalized 3D
    model verticies.
 */
std::vector<cv::Vec3f> getVertices();

/**
 *  Returns the total number of 3D model verticies.
 *
 *  @return  The total number of 3D model verticies.
 */
int getNumVertices();

/**
 *  Returns the index of the model. These indices should
     be
 *  unique and within [1,255] as they also define the
    rendering
 *  intensity within the common silhouette mask.
 *
 *  @return  The index of the 3D model.
 */
int getModelID();

/**
 *  Sets the index of the model. These indices should be
 *  unique and within [1,255] as they also define the
    rendering
 *  intensity within the common silhouette mask.
 *
 *  @param  The index of the 3D model.
 */
void setModelID(int i);

/**
 *  Sets the current pose to previously defined the
    initial pose
 *  and initialization state to false.
```

```
     */
    void reset();

private:
    int m_id;

    cv::Matx44f T_i;

    cv::Matx44f T_cm;

    cv::Matx44f T_n;

    bool initialized;

    bool hasNormals;

    std::vector<cv::Vec3f> vertices;
    std::vector<cv::Vec3f> normals;
    std::vector<GLuint> indices;
    std::vector<GLuint> offsets;

    QOpenGLBuffer vertexBuffer;
    QOpenGLBuffer normalBuffer;
    QOpenGLBuffer indexBuffer;

    bool buffersInitialsed;

    cv::Vec3f lbn;
    cv::Vec3f rtf;
    float scaling;


    /**
     *  Loads the model data from the specified file.
     *
     *  @param  objFilename The relative path to the OBJ/PLY
         file.
     */
    void loadModel(const std::string modelFilename);
};

#endif /* MODEL_H */



#ifndef OBJECT3D_H
#define OBJECT3D_H

#include "model.h"
```

```
class TCLCHistograms;
class TemplateView;

/**
 *  A representation of a 3D object that provides all
    nessecary information
 *  for region-based pose estimation using tclc-histograms.
    It extends the
 *  basic model class by including a set of tclc-histograms
    and a list of
 *  all corresponding templates used for pose detection.
 */
class Object3D : public Model
{
public:
    /**
     *  Constructor creating a 3D object class from a
        specified initial 6DOF pose, a
     *  scaling factor, a tracking quality threshhold and a
        set of distances to the
     *  camera for template generation used within pose
        detection. Here, also the set
     *  of n tclc-histograms is initialized, with n being
        the total number of 3D model
     *  vertices.
     *
     *  @param objFilename  The relative path to an OBJ/PLY
        file describing the model.
     *  @param tx  The models initial translation in X-
        direction relative to the camera.
     *  @param ty  The models initial translation in Y-
        direction relative to the camera.
     *  @param tz  The models initial translation in Z-
        direction relative to the camera.
     *  @param alpha  The models initial Euler angle
        rotation about X-axis of the camera.
     *  @param beta  The models initial Euler angle rotation
         about Y-axis of the camera.
     *  @param gamma  The models initial Euler angle
        rotation about Z-axis of the camera.
     *  @param scale  A scaling factor applied to the model
        in order change its size independent of the original
         data.
     *  @param qualityThreshold  The individual quality
        tracking quality threshold used to decide whether
        tracking and detection have been successful (should
        be within [0.5,0.6]).
```

```
     * @param templateDistances  A vector of absolute Z-
       distance values to be used for template generation (
       typically 3 values: a close, an intermediate and a
       far distance)
     */
    Object3D(const std::string objFilename, float tx, float
       ty, float tz, float alpha, float beta, float gamma,
       float scale, float qualityThreshold, std::vector<
       float> &templateDistances);

    ~Object3D();

    /**
     * Tells whether the tracking has been lost, i.e. the
       quality was below
     * the prescribed threshhold.
     *
     * @return  True if it has been lost and false
       otherwise.
     */
    bool isTrackingLost();

    /**
     * Sets the tracking state of the object.
     *
     * @param val True  if the tracking has been lost,
       false if it is currently being tracked successfully.
     */
    void setTrackingLost(bool val);

    /**
     * Returns the tracking quality threshold for this
       object used to decide
     * whether tracking and detection have been successful.
     *
     * @return  The tracking quality threshold.
     */
    float getQualityThreshold();

    /**
     * Returns the set of tclc-histograms associated with
       this object.
     *
     * @return  The set of tclc-histograms associated with
       this object.
     */
    TCLCHistograms *getTCLCHistograms();
```

```
    /**
     *  Generates all base and neighboring templates
        required for
     *  the pose detection algorithm after a tracking loss.
     *  Must be called after the rendering buffers of the
     *  corresponding 3D model have been initialized and
        while
     *  the offscreen rendering OpenGL context is active.
     */
    void generateTemplates();

    /**
     *  Returns the set of all pre-generated base and
        neighboring template views
     *  of this object used during pose detection.
     *
     *  @return  The set of all template views for this
        object.
     */
    std::vector<TemplateView*> getTemplateViews();

    /**
     *  Returns the number of Z-distances used during
        template view generation
     *  for this object.
     *
     *  @return  The number of Z-distances of the template
        views for this object.
     */
    int getNumDistances();

    /**
     *  Clears all tclc-histograms and resets the pose of
        the object to the initial
     *  configuration.
     */
    void reset();

private:
    bool trackingLost;

    float qualityThreshold;

    int numDistances;

    std::vector<float> templateDistances;

    std::vector<cv::Vec3f> baseIcosahedron;
```

```
    std::vector<cv::Vec3f> subdivIcosahedron;

    TCLCHistograms *tclcHistograms;

    std::vector<TemplateView*> baseTemplates;
    std::vector<TemplateView*> neighboringTemplates;

};


#endif /* OBJECT3D_H */



#ifndef OPTIMIZATION_ENGINE
#define OPTIMIZATION_ENGINE

#include <opencv2/core.hpp>
#include <opencv2/imgproc.hpp>
#include <opencv2/calib3d.hpp>

#include "rendering_engine.h"
#include "signed_distance_transform2d.h"
#include "tclc_histograms.h"
#include "object3d.h"

/**
 *  This class implements an iterative Gauss-Newton
    optimization strategy for
 *  minimizing the region-based cost function with respect
    to the 6DOF
 *  pose of multiple rigid 3D objects on the basis of tclc-
    histograms
 *  for pixel-wise posterior segmentation of a camera frame.
 */
class OptimizationEngine
{
public:
    /**
     *  Constructor of the optimization engine, that create
        a signed
     *  distance transform object for internal use.
     *
     *  @param width  The width in pixels of the camera
        frame at full resolution.
     *  @param height  The height in pixels of the camera
        frame at full resolution.
     */
    OptimizationEngine(int width, int height);
```

```
    ~OptimizationEngine();

    /**
     * Performs an hierachical iterative Gauss-Newton pose
       optimization
     * for multiple 3D objects based on a region-based cost
       fuction. The
     * implementation is parallelized on the CPU and uses
       the GPU only
     * for rendering the models with OpenGL. Given a coarse
       to fine image
     * pyramid (with at least 3 levels, created with a
       scaling factor of 2)
     * of the current camera frame, the poses of all
       provided 3D objects
     * that have been initialized beforehand will be
       refined.
     *
     * @param imagePyramid A coarse to fine image pyramid
       of the camera frame showing the objects in question
       (at least 3 levels, RGB, uchar).
     * @param objects A collection 3d objects of which the
       poses are supposed to be optimized.
     * @param runs A factor specifiyng how many times the
       default number of iterations per level are supposed
       to be performed (default = 1).
     */
    void minimize(std::vector<cv::Mat> &imagePyramid, std::
       vector<Object3D*> &objects, int runs = 1);

private:
    static OptimizationEngine *instance;

    RenderingEngine *renderingEngine;

    SignedDistanceTransform2D *SDT2D;

    int width;
    int height;

    void runIteration(std::vector<Object3D*> &objects, const
        std::vector<cv::Mat> &imagePyramid, int level);

    void parallel_computeJacobians(Object3D *object, const
       cv::Mat &frame, const cv::Mat &depth, const cv::Mat &
       depthInv, const cv::Mat &sdt, const cv::Mat &xyPos,
       const cv::Rect &roi, const cv::Mat &mask, int m_id,
```

```
        int level, cv::Matx66f &wJTJ, cv::Matx61f &JT, int
            threads);

    cv::Rect compute2DROI(Object3D *object, const cv::Size &
        maxSize, int offset);

    void applyStepGaussNewton(Object3D *object, const cv::
        Matx66f &wJTJ, const cv::Matx61f &JT);
};


/**
 *  This class extends the OpenCV ParallelLoopBody for
    efficiently parallelized
 *  computations. Within the corresponding for loop, the
    Jacobian terms required for
 *  the Gauss-Newton pose update step are computed for a
    single object.
 */
class Parallel_For_computeJacobiansGN: public cv::
   ParallelLoopBody
{
private:
    uchar *frameData, *maskData, *initializedData;

    float *histogramsFGData, *histogramsBGData, *sdtData, *
        depthData, *depthInvData, *K_invData;

    int *xyPosData;

    cv::Mat localFG, localBG;

    std::vector<cv::Point3i> centersIDs;

    int numHistograms, radius2, upscale, numBins, binShift,
        fullWidth, fullHeight, _m_id;

    float _fx, _fy, _zNear, _zFar;

    bool maskAvailable;

    cv::Rect _roi;

    cv::Matx33f K_inv;

    cv::Matx66f *_wJTJCollection;
    cv::Matx61f *_JTCollection;
```

```
    int _threads;

public:
    Parallel_For_computeJacobiansGN(TCLCHistograms *
       tclcHistograms, const cv::Mat &frame, const cv::Mat &
       sdt, const cv::Mat &xyPos, const cv::Mat &depth,
       const cv::Mat &depthInv, const cv::Matx33f &K, float
       zNear, float zFar, const cv::Rect &roi, const cv::Mat
        &mask, int m_id, int level, std::vector<cv::Matx66f>
        &wJTJCollection, std::vector<cv::Matx61f> &
       JTCollection, int threads);

    bool isOccluded (int idx, float dist, float d);


#endif //OPTIMIZATION_ENGINE



#ifndef POSE_ESTIMATOR6D_H
#define POSE_ESTIMATOR6D_H

#include <opencv2/core.hpp>
#include <opencv2/calib3d.hpp>
#include <opencv2/video.hpp>

#include "object3d.h"
#include "rendering_engine.h"
#include "optimization_engine.h"
#include "signed_distance_transform2d.h"
#include "template_view.h"

/**
 *  This class implements a region-based 6DOF pose estimator
     in form of a
 *  tracking and detection hybrid approach. It can estimate
    the poses of
 *  multiple rigid 3D objects based on given 3d models and
    the images of
 *  a single monocular RGB camera in real-time.
 */
class PoseEstimator6D
{
public:
    /**
     *  Constructor of the pose estimator initializing
        rectification
     *  maps for image undistorting and the rendering engine
        , given
```

```
     *  the 3x3 float intrinsic camera matrix
     *  K = [fx 0 cx]
     *      [0 fy cy]
     *      [0  0  1]
     *  and distortion coefficients (as needed by OpenCV).
     *  It also initializes the OpenGL rendering buffers for
         all
     *  provided 3D objects using the OpenGL context of the
        engine.
     *
     *  @param  width The width in pixels of the camera
        frame at full resolution.
     *  @param  height The height in pixels of the camera
        frame at full resolution.
     *  @param  zNear The distance of the OpenGL near plane.
     *  @param  zFar The distance of the OpenGL far plane.
     *  @param  K The intrinsic camera matrix.
     *  @param  distCoeffs The cameras lens distortion
        coefficients.
     *  @param  objects A collection of all 3D objects to be
         tracked.
     */
    PoseEstimator6D(int width, int height, float zNear,
       float zFar, const cv::Matx33f &K, const cv::Matx14f &
       distCoeffs, std::vector<Object3D*> &objects);

    ~PoseEstimator6D();

    /**
     *  Initializes/starts tracking based on the current
        camera frame
     *  for a specified 3D object using its initial pose by
        building
     *  the first set of tclc-histograms. If the object was
        already
     *  initialized this method will reset/stop tracking for
         it
     *  instead.
     *
     *  @param  frame The current camera frame  (RGB, uchar)
        .
     *  @param  objectIndex The index of the object to be
        initialized.
     *  @param  undistortFrame A flag indicating whether the
         image should first be undistorted for
        initialization (default = true).
     */
    void toggleTracking(cv::Mat &frame, int objectIndex,
```

```
        bool undistortFrame = true);

    /**
     *  This method tries to track and detect the 6DOF poses
         of all
     *  currently initialized rigid 3D objects by minimizing
         the
     *  region-based cost function using tclc-histograms.
        During
     *  successful tracking, the pose is estimated frame-to-
        frame.
     *  If tracking has been lost for an object, the pose
        will be
     *  estimated using a template matching approach for
        pose detection
     *  also based on tclc-histograms.
     *  Within this method, the 3D objectives are updated
        with the
     *  new estimated poses which can be obtained by calling
         getPose()
     *  on each object afterwards.
     *
     *  @param frame  The current camera frame (RGB, uchar).
     *  @param undistortFrame A flag indicating whether the
        image should first be undistorted for initialization
         (default = true).
     *  @param undistortFrame A flag indicating whether it
        should be checked for a tracking loss after pose
        estimation (default = true).
     */
    void estimatePoses(cv::Mat &frame, bool undistortFrame =
        true, bool checkForLoss = true);

    /**
     *  Resets/stops pose tracking for all objects by
        clearing the
     *  respective sets of tclc-histograms.
     */
    void reset();

private:
    int width;
    int height;

    cv::Matx33f K;
    cv::Matx14f distCoeffs;

    cv::Mat map1;
```

```
    cv::Mat map2;

    std::vector<Object3D*> objects;

    RenderingEngine *renderingEngine;
    OptimizationEngine *optimizationEngine;

    SignedDistanceTransform2D *SDT2D;

    cv::Mat lastFrame;

    bool initialized;

    int tmp;

    void relocalize(Object3D *object, std::vector<cv::Mat> &
       imagePyramid);

    cv::Rect computeBoundingBox(const std::vector<cv::
       Point3i> &centersIDs, int offset, int level, const cv
       ::Size &maxSize);

    float evaluateEnergyFunction(Object3D *object, const cv
       ::Mat &binned, int level, int threads);

    float evaluateEnergyFunction(Object3D *object, const cv
       ::Mat &mask, const cv::Mat &depth, const cv::Mat &
       binned, int level, int threads);

    float evaluateEnergyFunction(TCLCHistograms *
       tclcHistograms, const std::vector<cv::Point3i> &
       centersIDs, const cv::Mat &binned, const cv::Mat &
       heaviside, const cv::Rect &roi, int offsetX, int
       offsetY, int level, int threads);

    float evaluateEnergyFunction_local(TCLCHistograms *
       tclcHistograms, const std::vector<cv::Point3i> &
       centersIDs, const cv::Mat &binned, const cv::Mat &
       heaviside, const cv::Rect &roi, int offsetX, int
       offsetY, int level);

};

/**
 *  This class extends the OpenCV ParallelLoopBody for
    efficiently parallelized
 *  computations. Within the corresponding for loop, the
    region-based cost function is
```

```
 *  evaluated given the current camera image for a single
    object.
 */
class Parallel_For_evaluateEnergy: public cv::
   ParallelLoopBody
{
private:
    int* binsData;

    cv::Mat localFG;
    cv::Mat localBG;

    float* histogramsFGData;
    float* histogramsBGData;

    std::vector<cv::Point3i> _centersIDs;

    uchar *initializedData;

    int numHistograms;
    int radius;
    int radius2;

    int scale;

    int fullWidth;
    int fullHeight;

    int _offsetX;
    int _offsetY;

    float *hsData;

    cv::Rect _roi;

    float *_eCollection;

    int _threads;

public:
    Parallel_For_evaluateEnergy(TCLCHistograms *
       tclcHistograms, const std::vector<cv::Point3i> &
       centersIDs, const cv::Mat &bins, const cv::Mat&
       heaviside, const cv::Rect &roi, int offsetX, int
       offsetY, int level, cv::Mat &eCollection, int threads
       );

/**
```

```
 *  This class extends the OpenCV ParallelLoopBody for
    efficiently parallelized
 *  computations. Within the corresponding for loop, the RGB
     values per pixel
 *  of a color input image are converted to their
    corresponding histogram bin
 *  index.
 */
class Parallel_For_convertToBins: public cv::
   ParallelLoopBody
{
private:
    cv::Mat _frame;
    cv::Mat _binned;

    uchar *frameData;
    int *binnedData;

    int _numBins;

    int _binShift;

    int _threads;

public:
    Parallel_For_convertToBins(const cv::Mat &frame, cv::Mat
        &binned, int numBins, int threads);

/**
 *  This class extends the OpenCV ParallelLoopBody for
    efficiently parallelized
 *  computations. Within the corresponding for loop, for
    each pixel of a color
 *  input imagec (represented by the corrsponding histogram
    bin index) the
 *  average foreground and backgorund posterior probalility
    across a set of
 *  pre-computed tclc-histograms is computed. If that
    foregorund probablilty
 *  is greater than the background probability, the value of
     the pixel in the
 *  resulting posterior response map is set to 255 and 0
    otherwise.
 */
class Parallel_For_createPosteriorResponseMap: public cv::
   ParallelLoopBody
{
private:
```

```
    cv::Mat localFG;
    cv::Mat localBG;

    uchar *initializedData;

    int numHistograms;
    int numBins;

    cv::Mat _binned;
    cv::Mat _map;

    int *binnedData;
    uchar *mapData;

    int _threads;

public:
    Parallel_For_createPosteriorResponseMap(TCLCHistograms *
       tclcHistograms, const cv::Mat &binned, cv::Mat &map,
       int threads);

/**
 *  This class extends the OpenCV ParallelLoopBody for
    efficiently parallelized
 *  computations. It is the super class for parallelized
    template matching using
 *  either base templates or neighboring templates. Fot this
    , it provides an
 *  efficient method for cost function evaluation based on a
     compressed template
 *  representation.
 */
class Parallel_For_templateMatcher: public cv::
   ParallelLoopBody
{
public:

    float evaluateEnergyFunction(TCLCHistograms *
       tclcHistograms, const std::vector<PixelData> &
       compressedPixelData, const cv::Mat &binned, const cv
       ::Rect &roi, int offsetX, int offsetY);

/**
 *  This class extends the OpenCV ParallelLoopBody for
    efficiently parallelized
 *  computations. Within the corresponding for loop,
    tempalte matching for all
 *  base templates across the whole image is performed in a
```

```
    sliding window manner.
 *  This is accelerated by using a posterior response map to
     quickly detect regions
 *  where the cost functioin must not be evaluated.
 */
class Parallel_For_exhaustiveSearch: public
   Parallel_For_templateMatcher
{
private:
    Object3D *object;
    std::vector<TemplateView*> templateViews;

    cv::Mat binned;
    cv::Mat prMap;

    int level;
    int step;
    int diameter;

public:
    Parallel_For_exhaustiveSearch(Object3D *object, std::
       vector<TemplateView*> &templateViews, const cv::Mat &
       binned, const cv::Mat &prMap, int level, int step,
       int diameter);

    float computeMapMaskMatch(const cv::Mat &map, const cv::
       Mat &mask, int etaF, int offsetX, int offsetY, int
       innerOffset);

/**
 *  This class extends the OpenCV ParallelLoopBody for
    efficiently parallelized
 *  computations. Within the corresponding for loop,
    template matching is performed
 *  for all neighboring templates corresponding to one base
    template at the 2D location
 *  where this base template matched best at the lower image
     pyramid level during the
 *  previous exhaustive search.
 */
class Parallel_For_neighborSearch: public
   Parallel_For_templateMatcher
{
private:
    Object3D *object;
    TemplateView *templateView;
    std::vector<TemplateView*> neighbors;
```

```
    cv::Mat binned;

    int offsetX0;
    int offsetY0;

    int centerX0;
    int centerY0;

    int level;
    int levelDiff;

public:
    Parallel_For_neighborSearch(Object3D *object,
       TemplateView *templateView, const cv::Mat &binned,
       int level, int levelDiff);


#endif //POSE_ESTIMATOR6D_H


#ifndef RENDERING_ENGINE
#define RENDERING_ENGINE

#include <iostream>

#include <QOpenGLContext>
#include <QOffscreenSurface>

#include <QGLFramebufferObject>
#include <QOpenGLShaderProgram>
#include <QOpenGLFunctions_3_3_Core>

#include <opencv2/core.hpp>
#include <opencv2/imgproc.hpp>

#include "transformations.h"
#include "model.h"

/**
 *  This class implements an OpenGL-based offscreen
    rendering engine for generating
 *  images of projected 3D meshes based on given object
    poses and camera instrinsics.
 *  It supports one or mutiple objects to be rendered as
    binary masks, depth maps,
 *  normal maps or phong-shaded. It also allows to perform
    all renderings according
 *  to a specified image pyramid level at lower resolutions.
     The class is  implemented
```

```
 *  as a singleton.
 */
class RenderingEngine : public QOpenGLFunctions_3_3_Core
{
public:
    enum FrameType {
        MASK,
        RGB,
        RGB_32F,
        DEPTH
    };

    RenderingEngine(void);

    ~RenderingEngine(void);

    static RenderingEngine *Instance(void)
    {
        if (instance == NULL) instance = new RenderingEngine
           ();
        return instance;
    }

    /**
     *  Initializes the rendering engine instance given a 3
        x3 float
     *  intrinsic camera matrix
     *  K = [fx 0 cx]
     *      [0 fy cy]
     *      [0  0  1],
     *  a desired image resolution, a near and a far plane
        as well as the number
     *  of pyramidf level supported for the renderings.
     *
     *  @param  K The intrinsic camera matrix.
     *  @param  width The width in pixels of the rendered
        images at level 0.
     *  @param  height The height in pixels of the rendered
        images at level 0.
     *  @param  zNear The distance of the OpenGL near plane.
     *  @param  zFar The distance of the OpenGL far plane.
     *  @param  numLevels Number of supported pyramid levels
         with a downscale factor of 2.
     */
    void init(const cv::Matx33f &K, int width, int height,
       float zNear, float zFar, int numLevels);

    /**
```

```
     *  Returns the number of supported pyramid levels for
        rendering.
     *
     *  @return  The number of supported pyramid levels for
        rendering.
     */
    int getNumLevels();

    /**
     *  Sets a pyramid level to be used for rendering
        between 0 (full resolution)
     *  and getNumLevels() (the smallest resolution).
     *
     *  @param level The pyramid level to be used for
        rendering.
     */
    void setLevel(int level);

    /**
     *  Returns the current pyramid level used for rendering
        .
     *
     *  @return  The current pyramid level used for
        rendering.
     */
    int getLevel();

    /**
     *  Activates the OpenGL context of the rendering engine
        .
     */
    void makeCurrent();

    /**
     *  Deactivates the OpenGL context of the rendering
        engine.
     */
    void doneCurrent();

    /**
     *  Returns the OpenGL context of the rendering engine.
     *
     *  @return  The OpenGL context of the rendering engine.
     */
    QOpenGLContext *getContext();

    /**
     *  Returns the OpenGL ID of the frame buffer object
```

```
    used for offscreen rendering.
 *
 *  @return  The OpenGL ID of the frame buffer object
    used for offscreen rendering.
 */
GLuint getFrameBufferID();

/**
 *  Returns the OpenGL texture ID of the rendered color
    image.
 *
 *  @return  The OpenGL texture ID of the rendered color
     image.
 */
GLuint getColorTextureID();

/**
 *  Returns the OpenGL texture ID of the rendered depth
    buffer.
 *
 *  @return  The OpenGL texture ID of the rendered depth
     buffer.
 */
GLuint getDepthTextureID();

/**
 *  Returns the Z-distance of the near plane.
 *
 *  @return  The the Z-distance of the near plane.
 */
float getZNear();

/**
 *  Returns the Z-distance of the far plane.
 *
 *  @return  The the Z-distance of the far plane.
 */
float getZFar();

/**
 *  Returns a 4x4 float version of the intrinsic camera
    matrix wrt the current
 *  pyramid level
 *  K_4x4 = [fx/s 0 cx/s 0]
 *          [0 fy/s cy/s 0]
 *          [0  0    1   0]
 *          [0  0    0   1],
 *  with s = 1/2^level.
```

```
     *
     *  @return  A 4x4 float version of the intrinsic camera
         matrix wrt the current
     *  pyramid level.
     */
    cv::Matx44f getCalibrationMatrix();

    /**
     *  Renders a single model with a constant color and no
        shading in order to
     *  obtain a binary silhouette mask of it wrt its
        current pose.
     *
     *  @param model The model to be rendered.
     *  @param polyonMode The OpenGL polygon mode to be used
         (e.g. GL_FILL).
     *  @param invertDepth Whether to invert the depth test
        during rendering (default = false).
     *  @param r The red intensity of the model surface
        albedo in [0, 1] (default = 1.0).
     *  @param g The green intensity of the model surface
        albedo in [0, 1] (default = 1.0).
     *  @param b The blue intensity of the model surface
        albedo in [0, 1] (default = 1.0).
     *  @param drawAll Whether to draw the model even if it
        has not yet been initlaized for tracking (default =
        false).
     */
    void renderSilhouette(Model *model, GLenum polyonMode,
       bool invertDepth = false, float r = 1.0f, float g =
       1.0f, float b = 1.0f, bool drawAll = false);

    /**
     *  Renders a single model wrt its current pose using
        Phong shading.
     *
     *  @param model The model to be rendered.
     *  @param polyonMode The OpenGL polygon mode to be used
         (e.g. GL_FILL).
     *  @param r The red intensity of the model surface
        albedo in [0, 1] (default = 1.0).
     *  @param g The green intensity of the model surface
        albedo in [0, 1] (default = 0.5).
     *  @param b The blue intensity of the model surface
        albedo in [0, 1] (default = 0.0).
     *  @param drawAll Whether to draw the model even if it
        has not yet been initlaized for tracking (default =
        false).
```

```
 */
void renderShaded(Model *model, GLenum polyonMode, float
    r = 1.0f, float g = 0.5f, float b = 0.0f, bool
   drawAll = false);

/**
 *  Renders the per pixel surface normals of single
    model wrt its current pose where the
 *  normal direction is mapped from (x, y, z) in [-1, 1]
     to (r, g, b) in [0, 1].
 *
 *  @param model The model to be rendered.
 *  @param polyonMode The OpenGL polygon mode to be used
     (e.g. GL_FILL).
 *  @param drawAll Whether to draw the model even if it
    has not yet been initlaized for tracking (default =
    false).
 */
void renderNormals(Model *model, GLenum polyonMode, bool
    drawAll = false);

/**
 *  Renders a multiple models in a common scene with a
    constant color and no shading
 *  in order to obtain a their binary silhouette masks
    with correct occlusions
 *  according to their current poses. If no colors are
    spefified each model will by default
 *  get rendered with a constant color corresponding to
    their model index in the red channel.
 *
 *  @param model The models to be rendered.
 *  @param polyonMode The OpenGL polygon mode to be used
     (e.g. GL_FILL).
 *  @param invertDepth Whether to invert the depth test
    during rendering (default = false).
 *  @param colors A vector of colors to be used for each
     model (default = empty).
 *  @param drawAll Whether to draw all models even if
    they been not yet initlaized for tracking (default =
     false).
 */
void renderSilhouette(std::vector<Model*> models, GLenum
    polyonMode, bool invertDepth = false, const std::
   vector<cv::Point3f> &colors = std::vector<cv::Point3f
   >(), bool drawAll = false);

/**
```

```
     * Renders a multiple models in a common scene wrt
       their current poses using Phong shading.
     *
     * @param model The models to be rendered.
     * @param polyonMode The OpenGL polygon mode to be used
       (e.g. GL_FILL).
     * @param colors A vector of colors to be used for each
       model (default = empty).
     * @param drawAll Whether to draw the model even if it
       has not yet been initlaized for tracking (default =
       false).
     */
    void renderShaded(std::vector<Model*> models, GLenum
       polyonMode, const std::vector<cv::Point3f> &colors =
       std::vector<cv::Point3f>(), bool drawAll = false);

    /**
     * Renders the per pixel surface normals of multiple
       models in a common scene wrt their
     * current poses where the normal direction is mapped
       from (x, y, z) in [-1, 1] to (r, g, b)
     * in [0, 1].
     *
     * @param model The model to be rendered.
     * @param polyonMode The OpenGL polygon mode to be used
       (e.g. GL_FILL).
     * @param drawAll Whether to draw the model even if it
       has not yet been initlaized for tracking (default =
       false).
     */
    void renderNormals(std::vector<Model*> models, GLenum
       polyonMode, bool drawAll = false);

    /**
     * Projects the eight corners of a model's bouding box
       into the image and computes the
     * enclosing 2D bounding rect of these projections wrt
       the model's poae.
     *
     * @param model The model of which the bounding box is
       to be projected.
     * @param projections The resulting 2D coordinates of
       the projected bounding box corners.
     * @param boundingRect The resulting 2D bounding rect
       of the 2D projections.
     */
    void projectBoundingBox(Model *model, std::vector<cv::
       Point2f> &projections, cv::Rect &boundingRect);
```

```
    /**
     *  Downloads the most recently rendered image from the
        GPU to the host memory and converts
     *  it to an OpenCV image depending on a given frametype
        . Use MASK to obtain a silhouette
     *  mask image (single channel, uchar), RGB to obtain a
        color image (RGB, uchar), RGB_32F
     *  to obtain color image with normalized intensities in
         [0, 1] (RGB, float) or DEPTH to
     *  obtain the depth buffer.
     *
     *  @param type The frame type to be downloaded and
        returned (e.g. MASK, RGB, RGB32F or DEPTH).
     *
     *  @return  The most recently rendered image according
        to the desired frame type.
     */
    cv::Mat downloadFrame(RenderingEngine::FrameType type);

    /**
     *  Destroys and deletes the current rendering engine
        singleton instance.
     */
    void destroy();


private:
    static RenderingEngine *instance;

    int width;
    int height;

    int fullWidth;
    int fullHeight;

    float zNear;
    float zFar;

    int numLevels;

    int currentLevel;

    std::vector<cv::Matx44f> calibrationMatrices;
    cv::Matx44f projectionMatrix;
    cv::Matx44f lookAtMatrix;

    QOffscreenSurface *surface;
```

```
    QOpenGLContext *glContext;

    GLuint frameBufferID;
    GLuint colorTextureID;
    GLuint depthTextureID;

    int angle;

    cv::Vec3f lightPosition;

    QString shaderFolder;
    QOpenGLShaderProgram *silhouetteShaderProgram;
    QOpenGLShaderProgram *phongblinnShaderProgram;
    QOpenGLShaderProgram *normalsShaderProgram;

    bool initRenderingBuffers();

    bool initShaderProgram(QOpenGLShaderProgram *program,
       QString shaderName);

};


#endif //RENDERING_ENGINE



#ifndef SIGNED_DISTANCE_TRANSFORM2D_H
#define SIGNED_DISTANCE_TRANSFORM2D_H

#include <iostream>

#include <emmintrin.h>

#include <opencv2/core.hpp>

/**
 *  This class implements a signed 2D Euclidean distance
    transform
 *  of an arbitrary binary image (e.g. an object silhouette
    mask).
 *  Parts of this code are based on an open souce C-
    implementation
 *  available here:
 *  https://people.xiph.org/~tterribe/notes/edt.c
 *  It was originally published on this website:
 *  https://people.xiph.org/~tterribe/notes/edt.html
 *  However, the original code has been strongly modified,
    parallelized
```

```
 *  and extended, such that also the closest contour point
    is computed
 *  for each pixel in addition to its signed distance to it.
 */
class SignedDistanceTransform2D
{
public:
    /**
     *  Initializes the an instance with a specified maximum
         distance at wich the
     *  clostest contour points for every pixel are still
        computed.
     *
     *  @param  maxDist The maximal absolute distance at
        which the closest contour points are being comuted.
     */
    SignedDistanceTransform2D(float maxDist);

    ~SignedDistanceTransform2D();

    /**
     *  Computes the 2D Euclidean signed distance transform
        of a given input image as
     *  well as the coordinates of the clostest contour
        location for every pixel with
     *  CPU multi-threading.
     *
     *  @param  src The input image of which the distance
        transform shall be computed (single channel, float
        of uchar).
     *  @param  sdt The output 2D Euclidean signed distance
        transform of src.
     *  @param  xyPos The per pixel 2D coordinates of the
        closest contour points (two channel, integer).
     *  @param  threads The number of threads to be used for
         parallelization.
     *  @param  key In case of a uchar input image that is
        not binary, the value specidfies the intensitiy to
        be considered foregorund (default = 0, i.e. anything
         not equal to 0 is considered foreground).
     */
    void computeTransform(const cv::Mat &src, cv::Mat &sdt,
       cv::Mat &xyPos, int threads, uchar key = 0);

    /**
     *  Computes the first order derivatives of a given 2D
        Euclidean signed distance
     *  level-set in x- and y- direction at each pixel using
```

```
         central differences with
     *  CPU multi-threading.
     *
     *  @param  sdt The input 2D Euclidean signed distance
        transform of which the
     *  derivatives shall be computed (single channel, float
        ).
     *  @param  dX The output derivatives in x-direction (
        single channel, float).
     *  @param  dY The output derivatives in y-direction (
        single channel, float).
     *  @param  threads The number of threads to be used for
         parallelization.
     */
    void computeDerivatives(const cv::Mat &sdt, cv::Mat &dX,
        cv::Mat &dY, int threads);

private:
    float maxDist;
};


/**
 *  This class extends the OpenCV ParallelLoopBody for
    efficiently parallelized
 *  computations. Within the corresponding for loop, for
    every row in a binary input image
 *  the per pixel 1D signed distance transform is computed.
    Here, also the x locations of
 *  the closest contour points per pixel are calculated.
 */
template <class type>

class Parallel_For_distanceTransformRows: public cv::
   ParallelLoopBody
{
private:
    cv::Mat _src;
    cv::Mat _dd;
    cv::Mat _xPos;

    int *_v;
    int *_z;

    int _threads;

public:
    Parallel_For_distanceTransformRows(const cv::Mat &src,
```

```
        cv::Mat &dd, cv::Mat &xPos, int *v, int *z, int
        threads);

/**
 *  This class extends the OpenCV ParallelLoopBody for
    efficiently parallelized
 *  computations. Within the corresponding for loop, for
    every row in a input image
 *  the per pixel 1D signed distance transform is computed
    based on a given key value
 *  that specifies the intensitiy of the foreground region.
    Here, also the x locations of
 *  the closest contour points per pixel are calculated.
 */
class Parallel_For_distanceTransformRowsWithKey: public cv::
   ParallelLoopBody
{
private:
    cv::Mat _src;

    uchar _key;

    cv::Mat _dd;
    cv::Mat _xPos;

    int *_v;
    int *_z;

    int _threads;

public:
    Parallel_For_distanceTransformRowsWithKey(const cv::Mat
       &src, uchar key, cv::Mat &dd, cv::Mat &xPos, int *v,
       int *z, int threads);


/**
 *  This class extends the OpenCV ParallelLoopBody for
    efficiently parallelized
 *  computations. Within the corresponding for loop, the per
     pixel 2D signed distance
 *  transform is computed for every column based on the
    previously transformed rows.
 *  Here, also the 2D locations of the closest contour
    points per pixel are calculated.
 */
class Parallel_For_distanceTransformCols: public cv::
   ParallelLoopBody
```

```
{
private:
    cv::Mat _src;
    cv::Mat _dst;

    cv::Mat _xPos;
    cv::Mat _xyPos;

    int *_v;
    int *_z;
    int *_f;

    float _maxDist;

    int _threads;

public:
    Parallel_For_distanceTransformCols(const cv::Mat &src,
       cv::Mat &dst, const cv::Mat &xPos, cv::Mat &xyPos,
       float maxDist, int *v, int *z, int *f, int threads);


/**
 *  This class extends the OpenCV ParallelLoopBody for
    efficiently parallelized
 *  computations. Within the corresponding for loop, for
    each pixel the central differences
 *  in x-direction are computed from a given 2D Euclidean
    singed distance transform.
 */
template <class type>
class Parallel_For_distanceTransformDX: public cv::
   ParallelLoopBody
{
private:
    cv::Mat _sdt;
    cv::Mat _dX;

    type div;

    int stepSize;

    __m128 v_div;

    int _threads;

public:
    Parallel_For_distanceTransformDX(const cv::Mat &sdt, cv
```

```
        ::Mat &dX, int threads);


/**
 *  This class extends the OpenCV ParallelLoopBody for
    efficiently parallelized
 *  computations. Within the corresponding for loop, for
    each pixel the central differences
 *  in y-direction are computed from a given 2D Euclidean
    singed distance transform.
 */
template <class type>
class Parallel_For_distanceTransformDY: public cv::
   ParallelLoopBody
{
private:
    cv::Mat _sdt;
    cv::Mat _dY;

    type div;

    int stepSize;

    __m128 v_div;

    int _threads;

public:
    Parallel_For_distanceTransformDY(const cv::Mat &sdt, cv
       ::Mat &dY, int threads);

#endif //SIGNED_DISTANCE_TRANSFORM2D_H



#ifndef TCLC_HISTOGRAMS_H
#define TCLC_HISTOGRAMS_H

#include <opencv2/core.hpp>
#include <opencv2/imgproc.hpp>

class Model;

/**
 *  This class implements an statistical image segmentation
    model based on temporary
 *  consistent, local color histograms (tclc-histograms).
    Here, each histogram corresponds
 *  to a 3D vertex of a given 3D model.
```

```
 */
class TCLCHistograms
{
public:
    /**
     *  Constructor that allocates both normalized and not
        normalized foreground
     *  and background histograms for each vertex of the
        given 3D model.
     *
     *  @param  model The 3D model for which the histograms
        are being created.
     *  @param  numBins The number of bins per color channel
        .
     *  @param  radius The radius of the local image region
        in pixels used for updating the histograms.
     *  @param  offset The minimum distance between two
        projected histogram centers in pixels during an
        update.
     */
    TCLCHistograms(Model *model, int numBins, int radius,
       float offset);

    ~TCLCHistograms();

    /**
     *  Updates the histograms from a given camera frame by
        projecting all histogram
     *  centers into the image and selecting those close or
        on the object's contour.
     *
     *  @param  frame The color frame to be used for
        updating the histograms.
     *  @param  mask The corresponding binary shilhouette
        mask of the object.
     *  @param  depth The per pixel depth map of the object
        used to filter histograms on the back of the object,
     *  @param  K The camera's instrinsic matrix.
     *  @param  zNear The near plane used to render the
        depth map.
     *  @param  zFar The far plane used to render the depth
        map.
     */
    void update(const cv::Mat &frame, const cv::Mat &mask,
       const cv::Mat &depth, cv::Matx33f &K, float zNear,
       float zFar);

    /**
```

```
     * Computes updated center locations and IDs of all
       histograms that project onto or close
     * to the contour based on the current object pose at a
       specified image pyramid level.
     *
     * @param mask The binary shilhouette mask of the
       object.
     * @param depth The per pixel depth map of the object
       used to filter histograms on the back of the object,
     * @param K The camera's instrinsic matrix.
     * @param zNear The near plane used to render the
       depth map.
     * @param zFar The far plane used to render the depth
       map.
     * @param level The image pyramid level to be used for
       the update.
     */
    void updateCentersAndIds(const cv::Mat &mask, const cv::
       Mat &depth, const cv::Matx33f &K, float zNear, float
       zFar, int level);

    /**
     * Returns all normalized forground histograms in their
       current state.
     *
     * @return The normalized foreground histograms.
     */
    cv::Mat getLocalForegroundHistograms();

    /**
     * Returns all normalized background histograms in
       their current state.
     *
     * @return The normalized background histograms.
     */
    cv::Mat getLocalBackgroundHistograms();

    /**
     * Returns the locations and IDs of all histogram
       centers that where used for the last
     * update() or updateCentersAndIds() call.
     *
     * @return The list of all current center locations on
       or close to the contour and their corresponding IDs
       [(x_0, y_0, id_0), (x_1, y_1, id_1), ...].
     */
    std::vector<cv::Point3i> getCentersAndIDs();
```

```
/**
 *  Returns a 1D binary mask of all histograms where a
    '1' means that the histograms
 *  corresponding to the index has been intialized
    before.
 *
 *  @return A 1D binary mask telling wheter each
    histogram has been initialized or not.
 */
cv::Mat getInitialized();

/**
 *  Returns the number of histogram bin per image
    channel as specified in the constructor.
 *
 *  @return The number of histogram bins per channel.
 */
int getNumBins();

/**
 *  Returns the number of histograms, i.e. verticies of
    the corresponding 3D model.
 *
 *  @return The number of histograms.
 */
int getNumHistograms();

/**
 *  Returns the radius of the local image region in
    pixels used for updating the
 *  histograms as specified in the constructor.
 *
 *  @return The minumum distance between two projected
    histogram centers in pixels.
 */
int getRadius();

/**
 *  Returns the minumum distance between two projected
    histogram centers during an update
 *  as specified in the constructor.
 *
 *  @return The minumum distance between two projected
    histogram centers in pixels.
 */
float getOffset();

/**
```

```
     * Clears all histograms by resetting them to zero and
       setting their status to
     * uninitialized
     */
    void clear();


private:
    int numBins;

    int _numHistograms;

    int radius;

    float _offset;

    cv::Mat notNormalizedFG;
    cv::Mat notNormalizedBG;

    cv::Mat normalizedFG;
    cv::Mat normalizedBG;

    cv::Mat initialized;

    Model* _model;

    std::vector<cv::Point3i> _centersIDs;

    std::vector<cv::Point3i> computeLocalHistogramCenters(
       const cv::Mat &mask);

    std::vector<cv::Point3i>
       parallelComputeLocalHistogramCenters(const cv::Mat &
       mask, const cv::Mat &depth, const cv::Matx33f &K,
       float zNear, float zFar, int level);

    void filterHistogramCenters(int numHistograms, float
       offset);
};

/**
 * This class extends the OpenCV ParallelLoopBody for
    efficiently parallelized
 * computations. Within the corresponding for loop, for
    every projected histogram center on or
 * close to the object's contour, a new foreground and
    background color histogram are computed
 * within a local circular image region is computed using
```

```
    the Bresenham algorithm to scan the
 *  corresponding pixels.
 */
class Parallel_For_buildLocalHistograms: public cv::
   ParallelLoopBody
{
private:
    cv::Mat _frame;
    cv::Mat _mask;

    uchar* frameData;
    uchar* maskData;

    size_t frameStep;
    size_t maskStep;

    cv::Size size;

    std::vector<cv::Point3i> _centers;

    int _radius;

    int _numBins;

    int _binShift;

    int histogramSize;

    cv::Mat _sumsFB;

    int _m_id;

    int* localFGData;
    int* localBGData;

    int* _sumsFBData;

    int _threads;

public:
    Parallel_For_buildLocalHistograms(const cv::Mat &frame,
       const cv::Mat &mask, const std::vector<cv::Point3i> &
       centers, float radius, int numBins, cv::Mat &
       localHistogramsFG, cv::Mat &localHistogramsBG, cv::
       Mat &sumsFB, int m_id, int threads);

/**
 *  This class extends the OpenCV ParallelLoopBody for
```

```
    efficiently parallelized
 *  computations. Within the corresponding for loop, each
    previously computed local foreground
 *  and background color histogram is merged with their
    normalized temporally consistent
 *  representation based on respective learning rates.
 */
class Parallel_For_mergeLocalHistograms: public cv::
   ParallelLoopBody
{
private:
    int histogramSize;

    cv::Mat _sumsFB;

    int* notNormalizedFGData;
    int* notNormalizedBGData;

    float* normalizedFGData;
    float* normalizedBGData;

    uchar* initializedData;

    std::vector<cv::Point3i> _centersIds;

    float _alphaF;
    float _alphaB;

    int* _sumsFBData;

    int _threads;

public:
    Parallel_For_mergeLocalHistograms(const cv::Mat &
       notNormalizedFG, const cv::Mat &notNormalizedBG, cv::
       Mat &normalizedFG, cv::Mat &normalizedBG, cv::Mat &
       initialized, const std::vector<cv::Point3i>
       centersIds, const cv::Mat &sumsFB, float alphaF,
       float alphaB, int threads);

/**
 *  This class extends the OpenCV ParallelLoopBody for
    efficiently parallelized
 *  computations. Within the corresponding for loop, every 3
    D histogram center is projected
 *  into the image plane. Those that do not project on or
    close to the object's contour are
 *  being filtered based on a given binary silhouette mask
```

```
    and depth map at a specified image
 *  pyramid level.
 */
class Parallel_For_computeHistogramCenters: public cv::
   ParallelLoopBody
{
private:
    std::vector<cv::Vec3f> _verticies;

    std::vector<cv::Point3i>* _centersIds;

    cv::Mat _depth;
    cv::Mat _mask;

    uchar* maskData;

    cv::Matx44f _T_cm;
    cv::Matx33f _K;

    float _zNear;
    float _zFar;

    int _m_id;

    int _level;

    int downScale;
    int upScale;

    int _threads;

public:
    Parallel_For_computeHistogramCenters(const cv::Mat &mask
       , const cv::Mat &depth, const std::vector<cv::Vec3f>
       &verticies, const cv::Matx44f &T_cm, const cv::
       Matx33f &K, float zNear, float zFar, int m_id, int
       level, std::vector<cv::Point3i>* centersIds, int
       threads);

#endif /* TCLC_HISTOGRAMS_H */



#ifndef TEMPLATE_VIEW_H
#define TEMPLATE_VIEW_H

#include <vector>

#include <opencv2/core.hpp>
```

```
#include <opencv2/imgproc.hpp>

#include "rendering_engine.h"
#include "object3d.h"
#include "tclc_histograms.h"
#include "signed_distance_transform2d.h"

/**
 *  The template view data per pixel.
 */
struct PixelData
{
    // The original 2D pixel location.
    int x;
    int y;

    // The Heaviside value.
    float hsVal;

    // The number of tclc-histograms the pixel lies within.
    int ids_size;

    // The IDs of all tclc-histograms the pixel lies within.
    int* ids;
};

/**
 *  A class representing a single template view at multiple
    image scales
 *  for region-based object pose detection using tclc-
    histograms.
 */
class TemplateView {

public:
    /**
     *  Constructor for the template view at a given object
        rotation and
     *  distance to the camera.
     *
     *  @param  object The 3D object for which the template
        view is to be created.
     *  @param  alpha The Euler angle of the object's
        rotation around the x-axis (in degrees).
     *  @param  beta The Euler angle of the object's
        rotation around the y-axis (in degrees).
     *  @param  gamma The Euler angle of the object's
        rotation around the z-axis (in degrees).
```

```
     * @param distance The object's distance to the camera
         to be used.
     * @param numLevels Number of template pyramid levels
        to be created with a downscale factor of 2.
     * @param generateNeighbors A flag telling whether
        neighboring templates should also be created or not.
     */
    TemplateView(Object3D *object, float alpha, float beta,
       float gamma, float distance, int numLevels, bool
       generateNeighbors);

    ~TemplateView();

    /**
     *  Returns the 6DOF object pose coresponding to the
     *  template view in form of a 4x4 float matrix
     *  T_cm = [r11 r12 r13 tx]
     *         [r21 r22 r23 ty]
     *         [r31 r32 r33 tz]
     *         [  0   0   0  1],
     *  describing the transformation from model coordinates
         X_m
     *  into camera coordinates X_c.
     *
     *  @return The 6DOF object pose within the template.
     */
    cv::Matx44f getPose();

    /**
     *  Returns the Euler angle of the object's rotation
        around the
     *  x-axis coresponding to the template.
     *
     *  @return The object's rotation around the x-axis
        within the template (in degrees).
     */
    float getAlpha();

    /**
     *  Returns the Euler angle of the object's rotation
        around the
     *  y-axis coresponding to the template.
     *
     *  @return The object's rotation around the y-axis
        within the template (in degrees).
     */
    float getBeta();
```

```
/**
 *  Returns the Euler angle of the object's rotation
    around the
 *  z-axis coresponding to the template.
 *
 *  @return  The object's rotation around the z-axis
    within the template (in degrees).
 */
float getGamma();

/**
 *  Returns the object's distance to the camera
    coresponding to
 *  the template.
 *
 *  @return  The object's distance to the camera within
    the template.
 */
float getDistance();

/**
 *  Returns the total number of pixels in the object
    region at a given
 *  pyramid level.
 *
 *  @param level The pyramid level to be used.
 *  @return  The total number of pixels in the object
    region within the template.
 */
int getEtaF(int level);

/**
 *  Returns the binary mask image of the template at a
    given pyramid
 *  level.
 *
 *  @param level The pyramid level to be used.
 *  @return  The binary mask image of the template.
 */
cv::Mat getMask(int level);

/**
 *  Returns the 2D signed distance transform of the
    binary mask of the
 *  template at a given pyramid level.
 *
 *  @param level The pyramid level to be used.
 *  @return  The 2D signed distance transform of the
```

```
    binary mask of the template.
 */
cv::Mat getSDT(int level);

/**
 *  Returns the smoothed Heaviside representation of the
     2D signed distance
 *  transform of the template at a given pyramid level.
 *
 *  @param level The pyramid level to be used.
 *  @return  The smoothed Heaviside representation of
    the 2D signed distance transform of the template.
 */
cv::Mat getHeaviside(int level);

/**
 *  Returns the 2D region of interest around the object
    in the template at
 *  a given pyramid level.
 *
 *  @param level The pyramid level to be used.
 *  @return  The 2D region of interest around the object
     in the template.
 */
cv::Rect getROI(int level);

/**
 *  Returns the 2D (x, y) offset at which the template
    currently matches
 *  best with the corresponding matching score at a
    given pyramid level.
 *
 *  @param level The pyramid level to be used.
 *  @return  The 2D offset with the matching score (x, y
    , score).
 */
cv::Point3f getCurrentOffset(int level);

/**
 *  Sets the 2D (x, y) offset at which the template
    currently matches
 *  best with the corresponding matching score at a
    given pyramid level.
 *
 *  @param offset The 2D offset with the matching score
    (x, y, score).
 *  @param level The pyramid level to be used.
 */
```

```
    void setCurrentOffset(cv::Point3f &offset, int level);

    /**
     *  Returns the 2D centers and IDs of all tclc-
        histograms in the
     *  template at a given pyramid level.
     *
     *  @param level The pyramid level to be used.
     *  @return  The 2D centers and IDs of all tclc-
        histograms in the template [(x_0, y_0, id_0), (x_1,
        y_1, id_1), ...].
     */
    std::vector<cv::Point3i> getCentersAndIDs(int level);

    /**
     *  Returns a linearized representation of the template
        at a given pyramid
     *  level.
     *
     *  @param level The pyramid level to be used.
     *  @return  The linearized representation of the
        template.
     */
    std::vector<PixelData> &getCompressedPixelData(int level
       );

    /**
     *  Adds a neighboring template view to this template.
     *
     *  @param kv The neighboring template view to be added.
     */
    void addNeighborTemplate(TemplateView *kv);

    /**
     *  Returns the set of all neighboring templates of this
         template.
     *
     *  @return  The set of all neighboring templates of
        this template.
     */
    std::vector<TemplateView*> getNeighborTemplates();

private:
    RenderingEngine *renderingEngine;

    cv::Matx44f T_cm;

    std::vector<int> etaFPyramid;
```

```
    std::vector<cv::Mat> maskPyramid;
    std::vector<cv::Mat> sdtPyramid;
    std::vector<cv::Mat> heavisidePyramid;

    std::vector<cv::Rect> roiPyramid;

    std::vector<std::vector<cv::Point3i> > centersIDsPyramid
        ;

    std::vector<std::vector<PixelData> > pixelDataPyramid;

    cv::Point3f currentOffset;

    float _alpha;
    float _beta;
    float _gamma;

    float _distance;

    int _numLevels;

    std::vector<TemplateView*> neighbors;

    void compressTemplateData(const std::vector<cv::Point3i>
        &centersIDs, const cv::Mat &heaviside, const cv::
       Rect &roi, int radius, int level);

    cv::Rect computeBoundingBox(const std::vector<cv::
       Point3i> &centersIDs, int offset, int level, const cv
       ::Size &maxSize);
};


/**
 *  This class extends the OpenCV ParallelLoopBody for
    efficiently parallelized
 *  computations. Within the corresponding for loop, every
    pixel of a given 2D
 *  signed distance transform is mapped to its correspoding
    value in the
 *  smoothed Heaviside function representation.
 */
class Parallel_For_convertToHeaviside: public cv::
   ParallelLoopBody
{
private:
    cv::Mat _sdt, _heaviside;
```

```
    float *sdtData, *hsData;

    int _threads;

public:
    Parallel_For_convertToHeaviside(const cv::Mat &sdt, cv::
       Mat &heaviside, int threads)
    {
        _sdt = sdt;

        heaviside.create(_sdt.rows, _sdt.cols, CV_32FC1);
        _heaviside = heaviside;

        sdtData = _sdt.ptr<float>();
        hsData = _heaviside.ptr<float>();

        _threads = threads;
    }

    virtual void operator()( const cv::Range &r ) const
    {
        int range = _sdt.rows/_threads;

        int yEnd = r.end*range;
        if(r.end == _threads)
        {
            yEnd = _sdt.rows;
        }

        float s = 1.2f;

        for(int y = r.start*range; y < yEnd; y++)
        {
            float* sdtRow = sdtData + y*_sdt.cols;
            float* hsRow = hsData + y*_heaviside.cols;

            for(int x = 0; x < _sdt.cols; x++)
            {
                float dist = sdtRow[x];
                hsRow[x] = (fabs(dist) <= 8.0f) ? 1.0f/float
                   (CV_PI)*(-atan(dist*s)) + 0.5f : -1.0f;
            }
        }
    }
};
#endif /* TEMPLATE_VIEW_H */


    #ifndef TRANSFORMATIONS_H
```

```
#define TRANSFORMATIONS_H

#include <opencv2/core.hpp>
#include <opencv2/calib3d.hpp>

/**
 * A collection of geometric transformations implemented
    using the OpenCV matrix types.
 */
class Transformations
{

public:
    /**
     * Constructs a uniform 3D scale transform in 4x4
        homogeneous matrix
     * representation.
     *
     * @param s The uniform scale factor in all three
        dimensions.
     * @return A 4x4 homogenbeous 3D scale matrix.
     */
    static cv::Matx44f scaleMatrix(float s);

    /**
     * Constructs a general 3D scale transform in 4x4
        homogeneous matrix
     * representation.
     *
     * @param sx The scale factor in x-direction.
     * @param sy The scale factor in y-direction.
     * @param sz The scale factor in z-direction.
     * @return A 4x4 homogenbeous 3D scale matrix.
     */
    static cv::Matx44f scaleMatrix(float sx, float sy, float
        sz);

    /**
     * Constructs a 3D translation transform in 4x4
        homogeneous matrix
     * representation.
     *
     * @param tvec The 3D vector (tx, ty, tz) to be used as
         translation.
     * @return A 4x4 homogenbeous 3D translation matrix.
     */
    static cv::Matx44f translationMatrix(const cv::Vec3f
       tvec);
```

```
/**
 *  Constructs a 3D translation transform in 4x4
    homogeneous matrix
 *  representation.
 *
 *  @param tx Translation in x-direction.
 *  @param tx Translation in y-direction.
 *  @param tx Translation in z-direction.
 *  @return A 4x4 homogenbeous 3D translation matrix.
 */
static cv::Matx44f translationMatrix(float tx, float ty,
    float tz);

/**
 *  Constructs a 3D rotation transform in 4x4
    homogeneous matrix
 *  representation from a given axis and angle of
    rotation.
 *
 *  @param angle The angle of rotation.
 *  @param axis  The axis of rotation.
 *  @return A 4x4 homogenbeous 3D rotation matrix.
 */
static cv::Matx44f rotationMatrix(float angle, cv::Vec3f
    axis);

/**
 *  Constructs an OpenGL look-at transform in 4x4
    homogeneous matrix
 *  representation.
 *
 *  @param ex The x-coordinate of the eye/camera origin.
 *  @param ey The y-coordinate of the eye/camera origin.
 *  @param ez The z-coordinate of the eye/camera origin.
 *  @param cx The x-coordinate of the center point where
     the camera is looking at.
 *  @param cy The y-coordinate of the center point where
     the camera is looking at.
 *  @param cz The z-coordinate of the center point where
     the camera is looking at.
 *  @param ux The x-direction of the up-vector.
 *  @param uy The y-direction of the up-vector.
 *  @param uz The z-direction of the up-vector.
 *  @return A 4x4 homogenbeous look-at matrix.
 */
static cv::Matx44f lookAtMatrix(float ex, float ey,
   float ez, float cx, float cy, float cz, float ux,
```

```
    float uy, float uz);

/**
 *  Constructs a 4x4 OpenGL perspective projection
    matrix from a
 *  given field of view, aspect ration and a near and
    far plane.
 *
 *  @param fovy   The vertical field of view of the
    camera.
 *  @param aspect The aspect ratio of the image.
 *  @param near   The distance of the near clipping
    plane.
 *  @param far    The distance of the far clipping plane
    .
 *  @return A 4x4 perspective projection matrix.
 */
static cv::Matx44f perspectiveMatrix(float fovy, float
   aspect, float zNear, float zFar);

/**
 *  Constructs a 4x4 OpenGL perspective projection
    matrix from a
 *  given intrinsic matrix corresponding to a real
    camera.
 *
 *  @param K      A 3x3 intrinsic camera matrix.
 *  @param width  The width of the camer image in pixels
    .
 *  @param height The height of the camer image in
    pixels.
 *  @param near   The distance of the near clipping
    plane.
 *  @param far    The distance of the far clipping plane
    .
 *  @param flipY  A flag telling whether the direction
    of the y-axis of the camera is to flipped or not.
 *  @return A 4x4 perspective projection matrix matching
     the camera's intrinsics.
 */
static cv::Matx44f perspectiveMatrix(const cv::Matx33f&
   K, int width, int height, float zNear, float zFar,
   bool flipY = false);

/**
 *  Constructs 3x3 skew-symmetric matrix (also called
    axiator) from
 *  a given 3D vector.
```

```
     *
     *  @param a A 3D vector.
     *  @return The 3x3 skew-symmetric matrix corresponding
        to the 3D vector.
     */
    static cv::Matx33f axiator(cv::Vec3f a);

    /**
     *  Computes the exponential map from a given 6D vector
        of twist
     *  coordinates to the corresponding rigid body
        transform in 4x4
     *  homogeneous matrix representation.
     *
     *  @param xi A 6D vector of tiwst coordinates.
     *  @return A 4x4 homogenbeous rigid body transformation
         matrix corresponding to the twist coordinates.
     */
    static cv::Matx44f exp(cv::Matx61f xi);
};

#endif //TRANSFORMATIONS_H
```

A.1.2 Script Unity

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Runtime;
using UnityEngine.Windows.WebCam;

public class PhotoCapture : MonoBehaviour
{
    PhotoCapture photoCaptureObject = null;
    private TcpClient socketConnection;
    private Thread m_NetworkThread;
    private Matrix4x4 transfPose = new Matrix4x4();
    private float[] transfPose = new float[16];
    private readonly object transformLock = new object();
    private bool rcvd = false;

    // Use this for initialization
    void Start()
    {
        ConnectToTcpServer();
```

```
    targetTexture = new Texture2D(1280, 720,
        TextureFormat.RGBA32, false);

    PhotoCapture.CreateAsync(false, delegate (
        PhotoCapture captureObject) {
        photoCaptureObject = captureObject;

        CameraParameters c = new CameraParameters();
        c.cameraResolutionWidth = targetTexture.width;
        c.cameraResolutionHeight = targetTexture.height;
        c.pixelFormat = CapturePixelFormat.BGRA32;

        captureObject.StartPhotoModeAsync(c, delegate (
            PhotoCapture.PhotoCaptureResult result) {
            photoCaptureObject.TakePhotoAsync(
                OnCapturedPhotoToMemory);
        });
    });
}

private void ConnectToTcpServer()
{
    try
    {
        m_NetworkThread = new Thread(new ThreadStart(
            NetworkThread));
        m_NetworkThread.IsBackground = true;
        m_NetworkThread.Start();
    }
    catch (Exception e)
    {
        Debug.Log("On client connect exception: " + e);
    }
}



List<byte> imageBufferList = new List<byte>();
void OnCapturedPhotoToMemory(PhotoCapture.
    PhotoCaptureResult result, PhotoCaptureFrame
    photoCaptureFrame)
{
    // Copy the raw IMFMediaBuffer data into our empty
        byte list.
    photoCaptureFrame.CopyRawImageDataIntoBuffer(
        imageBufferList);
```

```
        // In this example, we captured the image using the
           BGRA32 format.
        // The raw image data will also be flipped so we
           access data in reverse.
        int stride = 4;
        float denominator = 1.0f / 255.0f;
        List<Color> colorArray = new List<Color>();
        for (int i = imageBufferList.Count - 1; i >= 0; i -=
            stride)
        {
            float a = (int)(imageBufferList[i - 0]) *
               denominator;
            float r = (int)(imageBufferList[i - 1]) *
               denominator;
            float g = (int)(imageBufferList[i - 2]) *
               denominator;
            float b = (int)(imageBufferList[i - 3]) *
               denominator;

            colorArray.Add(new Color(r, g, b, a));
        }

        // Take another photo
        photoCaptureObject.TakePhotoAsync(
           OnCapturedPhotoToMemory);
    }

    private void SendMessage()
    {
        m_Sending = false;
        bool success = false;
        try
        {
            NetworkStream stream = socketConnection.
               GetStream();
            if (stream.CanWrite)
            {
                stream.Write(imageBufferList.ToArray(), 0,
                   imageBufferList.Count);
                Debug.Log("Client sent his message - should
                   be received by server");
            }
        }
        catch
        {
            success = false;
            //client.client.Close();
        }
```

```
        m_Sending = !success;
    }

    private void sayHello()
    {
        NetworkStream stream = socketConnection.GetStream();
        if (stream.CanWrite)
        {
            //string clientMessage = "This is a message from
                one of your clients.\n";
            byte[] dataLength = BitConverter.GetBytes(
               cameraResolution.width * cameraResolution.
               height);
            stream.Write(dataLength, 0, sizeof(int));
            Debug.Log("Client sent his message - should be
               received by server");
        }
    }

    private void NetworkThread()
    {
        socketConnection = new TcpClient("192.168.1.193",
           27015);

        sayHello();

        // Get a stream object for reading

        using ( stream = socketConnection.GetStream())
        {
            int length;
            // Read incomming stream into byte arrary.

            while ((length = stream.Read(bytes, 0, bytes.
               Length)) != 0)
            {
                var incommingData = new byte[length];
                Array.Copy(bytes, 0, incommingData, 0,
                   length);
                for (int i = 0; i < 16; i++)
                    array[i] = System.BitConverter.toSingle(
                       incommingData + i * 4);

                lock(transformLock)
                {
                    for (int i = 0; i < 4; i++)
                        for (int j = 0; j < 4; j++)
```

```
                                transfPose[j][i] = array[i*4 + j
                                    ];

                            rcvd = true;
                        }
                    }
                }
            }

            void update()
            {
                if (rcvd)
                    lock(transformLock)
                    {
                        var matrix = transform.localToWorldMatrix;
                        matrix = matrix * transfPose;
                        transform.localPosition = matrix.GetColumn
                            (3);
                        transform.localScale = new Vector3(matrix.
                            GetColumn(0).magnitude, matrix.GetColumn
                            (1).magnitude, matrix.GetColumn(2).
                            magnitude);
                        float w = Mathf.Sqrt(1.0f + matrix[0,0] +
                            matrix[1,1] + matrix[2,2]) / 2.0f;
                        transform.localRotation = new Quaternion((
                            matrix[2,1]-matrix[1,2])/(4.0f*w), (
                            matrix[0,2] - matrix[2,0]) / (4.0f * w),
                            (matrix[1,0] - matrix[0,1]) / (4.0f * w),
                             w);

                        rcvd = false;
                    }
            }
        }
```

Bibliografia

[1] Alexander Krull, Eric Brachmann, Frank Michel, Michael Ying Yang, Stefan Gumhold, Carsten Rother. “Learning Analysis-by-Synthesis for 6D Pose Estimation in RGB-D Images”. In: (2015).

[2] Alexander Krull, Eric Brachmann, Sebastian Nowozin, Frank Michel, Jamie Shotton, Carsten Rother. “PoseAgent: Budget-Constrained 6D Object Pose Estimation via Reinforcement Learning”. In: (2016).

[3] Alvaro Collet, Dmitry Berenson, Siddhartha S. Srinivasa, Dave Ferguson. “Object recognition and full pose registration from a single image for robotic manipulation”. In: (2009).

[4] Andreas Doumanoglou, Rigas Kouskouridas, Sotiris Malassiotis, Tae-Kyun Kim. “Recovering 6D Object Pose and Predicting Next-Best-View in the Crowd”. In: (2015).

[5] Arsalan Mousavian, Dragomir Anguelov, John Flynn, Jana Kosecka. “3D Bounding Box Estimation Using Deep Learning and Geometry”. In: (2016).

[6] Bo Chen, Alvaro Parra, Jiewei Cao, Nan Li, Tat-Jun Chin. “End-to-End Learnable Geometric Vision by Backpropagating PnP Optimization”. In: (2020).

[7] Catherine Capellen, Max Schwarz, Sven Behnke. “ConvPoseCNN: Dense Convolutional 6D Object Pose Estimation”. In: (2020).

[8] Chen Song, Jiaru Song, Qixing Huang. “HybridPose: 6D Object Pose Estimation under Hybrid Representations”. In: (2020).

[9] Chi Li, Jin Bai, and Gregory D. Hager. “A Unified Framework for Multi-View Multi-Class Object Pose Estimation”. In: (2018).

[10] Chi Li, Jonathan Bohren, Eric Carlson, Gregory D. Hager. “Hierarchical Semantic Parsing for Object Pose Estimation in Densely Cluttered Scenes”. In: (2016).

[11] Dinh-Cuong Hoang , Achim J. Lilienthal , and Todor Stoyanov. “Panoptic 3D Mapping and Object Pose Estimation Using Adaptively Weighted Semantic Information”. In: (2020).

[12] Eric Brachmann, Alexander Krull, Frank Michel, Stefan Gumhold, Jamie Shotton, Carsten Rother. “Learning 6D Object Pose Estimation Using 3D Object Coordinates”. In: (2014).

[13] Eric Brachmann, Frank Michel, Alexander Krull, Michael Ying Yang, Stefan Gumhold, Carsten Rother. “Uncertainty-Driven 6D Pose Estimation of Objects and Scenes from a Single RGB Image”. In: (2016).

[14] Ge Gao, Mikko Lauri, Xiaolin Hu, Jianwei Zhang, Simone Frintrop. "CloudAAE: Learning 6D Object Pose Regression with On-line Data Synthesis on Point Clouds". In: (2021).

[15] Ge Gao, Mikko Lauri, Yulong Wang, Xiaolin Hu, Jianwei Zhang, Simone Frintrop. "6D Object Pose Regression via Supervised Learning on Point Clouds". In: (2020).

[16] Gu Wang, Xiangyang Ji, Fabian Manhardt, Jianzhun Shao, Nassir Navab, Federico Tombari. "Self6D: Self-Supervised Monocular 6D Object Pose Estimation". In: (2020).

[17] Gu Wang1, Fabian Manhardt, Federico Tombari, Xiangyang Ji. "GDR-Net: Geometry-Guided Direct Regression Network for Monocular 6D Object Pose Estimation". In: (2021).

[18] Hans Schwarz, Hannes Schulz, Sven Behnke. "RGB-D Object Recognition and Pose Estimation based on Pre-trained Convolutional Neural Network Features". In: (2015).

[19] Henning Tjaden, Ulrich Schwanecke, Elmar Schomer, Daniel Cremers. "A Region-based Gauss-Newton Approach to Real-Time Monocular Multiple Object Tracking". In: *IEEE transactions on pattern analysis and machine intelligence* (2019).

[20] Jonathan Tremblay, Thang To, Balakumar Sundaralingam, Yu Xiang, Dieter Fox, Stan Birchfield. "Deep Object Pose Estimation for Semantic Robotic Grasping of Household Objects". In: (2018).

[21] Juil Sock, S.Hamidreza Kasaei, Luis Seabra Lopes, Tae-Kyun Kim. "Multi-view 6D Object Pose Estimation and Camera Motion Planning using RGBD Images". In: (2017).

[22] Mahdi Rad, Vincent Lepetit. "BB8: A Scalable, Accurate, Robust to Partial Occlusion Method for Predicting the 3D Poses of Challenging Objects without Using Depth". In: (2017).

[23] Manuel Martinez, Alvaro Collet, Siddhartha S. Srinivasa. "MOPED: A scalable and low latency object recognition and pose estimation system". In: (2010).

[24] Mingliang Fu, Weijia Zhou. "DeepHMap++: Combined Projection Grouping and Correspondence Learning for Full DoF Pose Estimation". In: (2019).

[25] Sabera Hoque, Md. Yasir Arafat, Shuxiang Xu, Anada Maiti, Yuchen Wei. "A Comprehensive Review on 3D Object Detection and 6D Pose Estimation with Deep Learning". In: *IEEE Access, Volume 9* (2021).

[26] Sergey Zakharov, Ivan Shugurov, Slobodan Ilic. "DPOD: 6D Pose Object Detector and Refiner". In: (2019).

[27] Stefan Hinterstoisser, Stefan Holzer, Cedric Cagniart, Slobodan Ilic, Kurt Konolige, Nassir Navab, Vincent Lepetit. "Multimodal Templates for Real-Time Detection of Texture-less Objects in Heavily Cluttered Scenes". In: (2012).

[28] Thanh-Toan Do, Ming Cai, Trung Pham, Ian Reid. "Deep-6DPose: Recovering 6D Object Pose from a Single RGB Image". In: (2018).

[29] Tomáš Hodaň, Xenophon Zabulis, Manolis Lourakis, Štěpán Obdržálek1, Jiří Matas. "Detection and Fine 3D Pose Estimation of Texture-less Objects in RGB-D Images". In: (2015).

[30] Victor A. Prisacariu, Ian D. Reid. "PWP3D: Real-Time Segmentation and Tracking of 3D Objects". In: (2010).

[31] Wadim Kehl, Fabian Manhardt, Federico Tombari, Slobodan Ilic, Nassir Navab. "SSD-6D: Making RGB-Based 3D Detection and 6D Pose Estimation Great Again". In: (2017).

[32] Wei Chen, Xi Jia1 Hyung, Jin Chang, Jinming Duan, Ales Leonardis. "G2L-Net: Global to Local Network for Real-time 6D Pose Estimation with Embedding Vector Features". In: (2020).

[33] Xingyu Liu, Shun Iwase, Kris M. Kitani. "KDFNet: Learning Keypoint Distance Field for 6D Object Pose Estimation". In: (2021).

[34] Yan Di, Fabian Manhardt, Gu Wang, Xiangyang Ji, Nassir Navab, Federico Tombari. "SO-Pose: Exploiting Self-Occlusion for Direct 6D Pose Estimation". In: (2021).

[35] Yi Li, Gu Wang, Xiangyang Ji, Yu Xiang, Dieter Fox. "DeepIM: Deep Iterative Matching for 6D Pose Estimation". In: (2019).

[36] Yifei Shi, Junwen Huang, Xin Xu, Yifan Zhang, Kai Xu. "StablePose: Learning 6D Object Poses from Geometrically Stable Patches". In: (2021).

[37] Yingzhao Zhu, Man Li, Wensheng Yao, Chunhua Chen. "A Review of 6D Object Pose Estimation". In: *IEEE ITAIC, 10th* (2022).

[38] Yisheng He, Wei Sun, Haibin Huang, Jianran Liu, Haoqiang Fan, Jian Sun. "PVN3D: A Deep Point-wise 3D Keypoints Voting Network for 6DoF Pose Estimation". In: (2020).

[39] Yu Xiang, Tanner Schmidt, Venkatraman Narayanan, Dieter Fox. "PoseCNN: A Convolutional Neural Network for 6D Object Pose Estimation in Cluttered Scenes". In: (2017).

[40] Zongxin Yang, Xin Yu, Yi Yang. "DSC-PoseNet: Learning 6DoF Object Pose Estimation via Dual-scale Consistency". In: (2021).

Ringraziamenti

In calce al presente lavoro, desidero ringraziare diverse persone, senza le quali questa tesi non esisterebbe nemmeno.

Ringrazio il relatore del progetto, Andrea Sanna, che in questo lungo periodo di lavoro mi ha guidato con le sue indicazioni e suggerimenti per la formulazione dell'elaborato.

Più di tutti ringrazio i miei genitori ed il nonno Giacomo. Grazie per avuto fede ed avermi permesso di portare a termine il ciclo universitario attraverso innumerevoli contrattempi.

Un ringraziamento particolare va alla mia fidanzata Roberta che non ha smesso di crederci e mi ha sempre aiutato a vedere il risvolto positivo.

Infine, spero che questo piccolo traguardo sia l'inizio di una nuova fase della mia vita ed un buon auspicio per la carriera appena avviata.