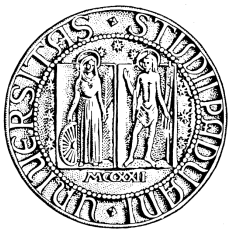

UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

TESINA

APPLICAZIONI DI REALTÀ AUMENTATA: QCAR SDK DI QUALCOMM

REALTORE: Dott. Fantozzi Carlo

LAUREANDO: Senigaglia Marco

A.A. 2011-2012

Ai miei genitori Giuliana e Umberto,
per il loro sostegno.

Indice

Capitolo 1 La realtà aumentata

La realtà aumentata (AR, dall'inglese Augmented Reality) è una branca della computer graphics che studia e sviluppa sistemi in grado di combinare immagini provenienti dal mondo reale con informazioni e oggetti calcolati da computer. L'utente di un'applicazione di AR, utilizzando opportune apparecchiature, è nella condizione di vivere un'esperienza sensoriale arricchita di informazioni ed elementi virtuali, a volte anche interagendo con loro.

1.1 Introduzione

In questa tesina si vedranno gli aspetti generali della realtà aumentata e successivamente lo sviluppo e l'analisi di un'applicazione utilizzando un particolare strumento, QCAR SDK, progettato per la piattaforma Android. Nel dettaglio, il contenuto dei capitoli della tesina è il seguente.

- Capitolo 1: storia della realtà aumentata, funzionamento di un sistema AR, tracciamento e applicazioni.
- Capitolo 2: analisi matematica del tracciamento ottico utilizzato anche da QCAR SDK.
- Capitolo 3: studio di QCAR SDK e analisi dettagliata delle sue funzioni principali.
- Capitolo 4: creazione di un'applicazione di AR con QCAR SDK e analisi del codice sviluppato.

1.2 Cenni Storici

Nel secolo scorso, con l'avvento dei primi computer e il conseguente avanzamento delle tecniche di comunicazione, andò a svilupparsi l'idea di poter creare una realtà alternativa nella quale l'utente potesse immergersi con tutti i sensi. Il primo a creare

una macchina che permettesse questa totale immersione in un altro mondo fu Morton Heiling che tra il 1957 e il 1962 creò e brevettò un simulatore chiamato *Sensorama* che, durante la proiezione di un film, coinvolgeva tutti i sensi. Solamente nel 1989 Jaron Lanier, fondatore della *VPL Research*, coniò il termine Realtà Virtuale (VR, dall'inglese Virtual Reality). Comunque già negli anni '60-'70 ci fu un forte sviluppo della VR, con la creazione di dispositivi tali da facilitarne l'uso e l'immersione per l'utente; infatti nel 1966 Ivan Sautherland inventò l'head-mounted display (HMD) (Figura 1.2.1), cioè un visore montato su speciali occhiali o su un casco, mentre nel 1975 Myron Krueger creò un ambiente chiamato *Videoplace* che dava la possibilità di interagire con il mondo virtuale.

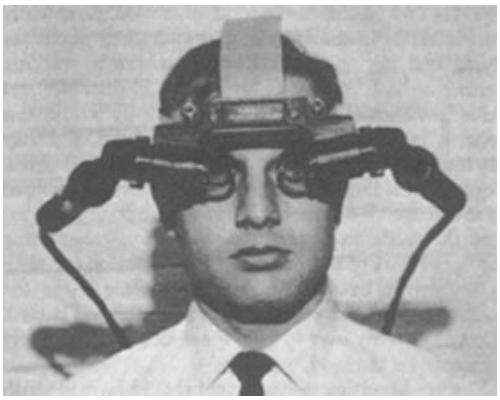

Figura 1.2.1*: Ivan Sautherland - Primo Head Mounted Display*

Il termine “Realtà Aumentata” venne coniato “casualmente” da Tom Caudell, quando nel 1990 durante il cablaggio di aeromobili Boing si avvalse dell'aiuto di un sistema che in tempo reale gli desse informazioni su come installare i cavi senza commettere errori (v. Figura 1.2.2).

Figura 1.2.2: Cablaggio assistito di un aeromobile Boing

Da allora la differenza tra Realtà Aumentata e Realtà Virtuale è fonte di dibattito. Nel corso degli anni gli esperti di computer graphics hanno fornito diverse definizioni formali dalle quali si evince che le due discipline, pur attingendo allo stesso bagaglio di conoscenze, sono concettualmente diverse. In sintesi si può affermare che mentre la VR riguarda l'immersione in un mondo artificiale, la AR si interessa di rendere la realtà più significativa sovrapponendo elementi virtuali strettamente correlati con il mondo reale stesso.

1.3 Sistema AR

Il corretto funzionamento di un'applicazione di AR si ha quando la combinazione di immagini provenienti dal mondo reale e gli oggetti virtuali risultano sincronizzati e la loro combinazione risulta corretta. Quindi si può affermare che il problema fondamentale della AR sia quello della corretta combinazione tra i mondi reale e virtuale; questo interessa tecnologie differenti in funzione del contesto in cui l'applicazione deve essere utilizzata.

L'hardware e il software che permettono lo sviluppo di applicazioni di realtà aumentata sono i più diversi quindi risulta difficile classificarli differenziandoli per una delle due categorie. Tuttavia, qualsiasi sistema di realtà aumentata può essere specificato da un comune schema generale.

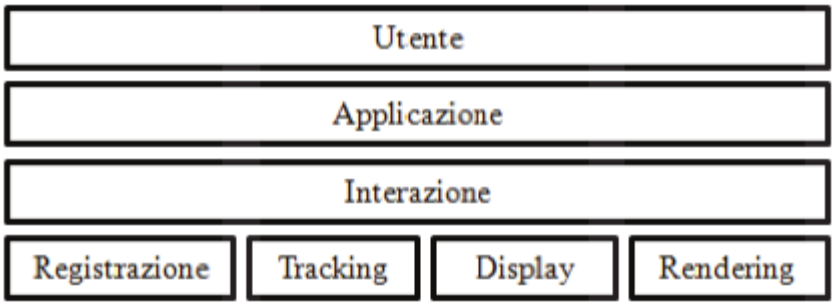

Figura 1.3.1: Schema a livelli di un sistema di realtà aumentata

Seguendo lo schema di Figura 1.3.1 a livello più basso abbiamo i processi base di *Tracking, Registrazione, Visualizzazione e Rendering* che vengono di seguito illustrati.

Il *Tracking* si occupa dell'individuazione della posizione dell'osservatore rispetto alla scena. In altri termini tale processo fornisce in tempo reale la posizione del punto di vista dell'osservatore rispetto ad un sistema di riferimento globale assegnato per convenzione all'ambiente reale in cui l'osservatore si trova. In altri casi il tracciamento consiste nella localizzazione degli oggetti nella scena reale rispetto alla posizione dell'osservatore. Esistono sistemi di tracking di diverso genere: i più importanti verranno discussi nel prossimo paragrafo.

La *registrazione* è il processo che si occupa di allineare gli oggetti virtuali al punto di vista dell'osservatore sulla scena, applicando le opportune trasformazioni geometriche.

Accanto alla registrazione e al tracking c'è il processo di *Visualizzazione*. Per tale funzionalità i dispositivi di visualizzazione (display) sono il supporto principale. La tipologia di dispositivo dominante è quella degli *Head Mounted Display*, i quali consentono una elevata mobilità e un veloce allineamento del punto di vista dell'osservatore con la scena proiettata, anche se stanno guadagnando sempre più spazio sulla scena degli apparecchi per la visualizzazione i dispositivi mobili con schermi ad alta definizione quali smartphone, cellulari o tablet.

Il terzo processo base è il *rendering*, che riguarda la sovrapposizione degli elementi virtuali alle immagini reali. Le proprietà che caratterizzano tale processo sono la velocità di aggiornamento delle immagini prodotte e la capacità di produrre immagini fotorealistiche. Velocità di aggiornamento e qualità dell'immagine prodotta sono proprietà in opposizione, infatti cercare un sistema efficiente per la velocità di

aggiornamento dell'immagine prodotta significa rinunciare a tecniche di rendering più sofisticate con un conseguente peggioramento della qualità delle immagini da sovrapporre.

Passando dal livello più basso dei processi delle applicazioni AR (v. Figura 1.3.1) al livello superiore si arriva al processo interattivo, ovvero un sistema che dia la possibilità di interagire con gli elementi virtuali. Tale processo, ove presente, necessita di strumentazione sviluppata ad hoc. In tal senso sensori, sistemi di tracciamento della posizione e accelerometri consento di estendere l'interazione con la scena anche ad altri sensi, oltre a quello visivo, e come sarà illustrato nelle sezioni successive anche di creare e manipolare oggetti virtuali puntando agli oggetti reali presenti nella scena.

1.3.1 Tracciamento

La corretta sovrapposizione di oggetti virtuali a immagini reali è uno dei requisiti fondamentali per le applicazioni di AR, come spiegato precedentemente. Gli elementi virtuali devono essere accuratamente sovrimposti e scalati in tutte le loro dimensioni. Per alcune applicazioni questo è fondamentale: infatti mentre la sovrapposizione non corretta dei frame virtuali e reali in un'applicazione di intrattenimento risulta poco efficace e sgradevole, lo stesso difetto avrebbe conseguenze disastrose durante operazioni chirurgiche praticate con il supporto di applicazioni di AR. In questo ambito è fondamentale non solo disporre di una corretta registrazione quando l'utente è fermo, ma anche quando si trova in movimento rispetto alla scena. A tale scopo è indispensabile che sia l'orientamento che la posizione dell'osservatore siano rilevate in tempo reale; nel caso specifico devono essere rilevati l'orientamento e la posizione della testa.

Inoltre il problema del tracking, in un applicazione di AR che preveda l'interazione con la scena, si estende anche ai dispositivi di puntamento.

Per il processo di tracciamento si distinguono due categorie di sistemi:

- *outside-in*;
- *inside-out.*

La distinzione tra le due categorie è basata sulla configurazione di sensori ed emettitori rispetto agli oggetti da puntare. I sistemi di tipo o*utside-in* hanno i loro sensori disposti in posizioni fisse all'interno della scena; gli oggetti da seguire sono

forniti di emettitori e marker. I sistemi di tipo i*nside-out* utilizzano sensori installati direttamente sugli oggetti da seguire.

Per un buon funzionamento del tracking nei sistemi AR, devono essere soddisfatti almeno i seguenti requisiti:

- accuratezza di misura, sia per l'inclinazione che per la posizione;
- sufficiente velocità di acquisizione (minimo 30 Hz);
- buona mobilità dell'utente(evitare il più possibile cavi o spazi di lavoro ristretti).

Nello sviluppare i sistemi di tracking sono stati seguiti diversi approcci, ognuno dei quali presenta caratteristiche positive e negative in funzione dell'utilizzo previsto. Una buona strategia consiste nell'integrare sistemi diversi affinché i limiti di un sistema vengano compensati dalle qualità di un altro.

Di seguito vengono presentati quattro approcci diversi al problema del tracking:

- il tracking inerziale;
- il tracking acustico;
- il tracking magnetico;
- il tracking ottico;

Il tracking inerziale utilizza giroscopi e accelerometri per determinare la posizione e l'orientamento nello spazio dei dispositivi di misura. Le misure derivanti da tali dispositivi non forniscono una misura diretta della posizione e dell'orientamento ma necessitano di una fase di elaborazione. Le misure delle accelerazioni devono essere integrate due volte per ricavare i parametri cinematici dell'oggetto da tracciare. Tale processo è soggetto al problema della deriva, che conduce a frequenti calibrazioni dell'apparecchiatura. Una caratteristica positiva di questo metodo di tracciamento è che non viene influenzato da eventuali interferenze magnetiche; inoltre esso non pone restrizioni sulle dimensioni dell'ambiente di funzionamento, anche se non risulta molto accurato per piccoli spostamenti.

I sistemi di tracciamento acustici si basano sull'emissione di onde sonore: indicativamente hanno lo stesso funzionamento dei radar. Se viene utilizzata una sola coppia trasmettitore-ricevitore, si riesce a rilevare la distanza tra il punto da tracciare e un punto fisso. Per ottenere i dati relativi alla posizione nello spazio è necessario avere un trasmettitore e tre ricevitori o viceversa, cioè, tre trasmettitori ed un

ricevitore. In questo tipo di sistemi la distanza viene calcolata utilizzando la lunghezza d'onda e i dati rilevati inerenti al tempo di risposta. I pregi di questi sistemi sono la leggerezza e l'indipendenza da disturbi dovuti ad interferenze magnetiche, essendo le onde acustiche immuni da tali problematiche. Al contrario questi dispositivi subiscono le interferenze acustiche. Inoltre l'accuratezza delle misure è correlata alle condizioni ambientali, essendo la velocità di propagazione delle onde acustiche dipendente dalla densità dell'aria nella quale viaggiano.
Un ulteriore difetto non poco importante è l'impossibilità di avere una rilevazione dei dati precisa se tra trasmettitore e ricevitore si interpone un qualsiasi oggetto, riducendone così l'affidabilità.
Il tracking magnetico si basa sulla misura del campo magnetico e utilizza sia onde a bassa frequenza che pulsate. È composto da un ricevitore e da un trasmettitore. I dati delle misure vengono elaborati per calcolare la posizione e l'orientamento del ricevitore rispetto all'emettitore. A differenza di altri dispositivi di tracciamento, quello magnetico non subisce i problemi di occlusione emettitore-ricevitore. I principali svantaggi connessi all'impiego di questa tecnologia riguardano la sensibilità alle interferenze magnetiche causate da onde radio e superfici metalliche. Infine l'accuratezza di misura diminuisce con l'aumentare della distanza tra emettitore e ricevitore.
L'intensità luminosa viene sfruttata dagli apparecchi di tracking ottici per calcolare la posizione degli oggetti nello spazio reale. Ogni punto del mondo reale riflette o genera luce, sfruttando la legge fisica per cui l'energia del raggio di luce diminuisce con il quadrato della distanza, si può stimare la posizione del ricevitore/emettitore misurando tale energia. Come ricevitori si possono utilizzare sensori CCD (*Charged Coupled Device*) per il calcolo della posizione di elementi passivi o fotodiodi laterali per la determinazione di elementi attivi. Esempi tipici di dispositivi passivi sono i marker e di dispositivi attivi i led. Nella categoria dei sistemi di puntamento ottici rientrano i sistemi di riconoscimento dell'immagine che utilizzano algoritmi di grafica computazionale per elaborare immagini, contenenti gli oggetti da seguire, e calcolarne la loro posizione rispetto al punto di vista della camera sulla scena. Il sistemi di tracking di tipo ottico assicurano velocità di elaborazione adeguate e non pongono limiti sullo spazio di osservazione. Per contro sono sensibili alla maggiore o

minore visibilità degli oggetti da osservare e all'intensità della luce.
L'analisi dei dispositivi di tracking finora svolta evidenzia, per ciascun sistema, limiti e qualità. Una scelta vincente potrebbe essere quella di combinare diversi dispositivi in modo tale che le caratteristiche positive di ciascuna tecnica migliorino e rendano le mancanze degli altri metodi ininfluenti nel calcolo finale del tracciamento. Sebbene questa appaia una tattica vincente, l'interazione di più strumenti accresce la complessità dei sistemi e inevitabilmente rende più complicata la gestione e il controllo dell'applicazione. Inoltre le tecnologie illustrate sono state sviluppate per ambienti chiusi, ovvero per ambienti limitati facilmente controllabili.
L'utilizzo dei sistemi di tracking per applicazioni in ambienti aperti avviene in una situazione diametralmente opposta: gli ambienti aperti sono fuori controllo e richiedono la mobilità della strumentazione. Infine è necessario considerare il processo di tracking per applicazioni di AR anche nell'ambito del design collaborativo, dove ciascun utente deve poter interagire autonomamente con lo scenario condiviso. In questo ambito occorre sviluppare applicazioni server in grado di gestire coerentemente il processo di tracking per più utenti e strumenti.
Per il tracking in ambienti aperti non si utilizzano marker o sensori installati sul dispositivo o sulla scena, bensì si usa una tracciamento tramite GPS (Global Positioning System) per individuare il punto in cui si trova l'utente e si utilizzano sensori quali giroscopio e bussola digitale per capire l'orientamento della camera.
Stabilita la posizione e la direzione dell'utente si scaricano da una connessione Internet i punti di interesse, sempre calcolati tramite GPS per luoghi aperti molto ampi, altrimenti tramite riconoscimento delle immagini.

1.3.2 I display

I sistemi di visualizzazione finora sviluppati per le applicazioni di AR sono raggruppabili in tre categorie.

- *Head mounted display*: sono dispositivi indossati direttamente sulla testa dall'osservatore, come occhiali o caschi.
- *Hand held displays*: sono i visualizzatori da tenere in mano, tipo palmari e cellulari.
- *Spatial displays*: sono dei dispositivi fisici sui quali vengono proiettate le immagini, come gli ologrammi che spesso vengono utilizzati per applicazioni

destinate ai musei.

Gli *Head Mounted Display* sono i visori più efficaci e pratici, grazie alla sensazione di immersione che garantiscono all'utente, e sono indicati con l'acronimo *HMD*. Per questi dispositivi si distinguono tre metodi differenti di implementazione in funzione dei dispositivi impiegati per la visualizzazione della scena. I display che utilizzano piccoli visori ottici sono denominati *Head Mounted Projector* e sono di due tipi:

- *optical See Through*
- *video See Through.*

Rientrano nella categoria anche i cosidetti *retinal display* che utilizzano i laser per generare i contenuti sintetici da visualizzare.

I dispositivi Optical See Through (v. Figura 1.3.2) utilizzano un visore di fascio ottico, consistente in uno specchio traslucido che trasmette la luce in una direzione e contemporaneamente la riflette nell'altra. Si basa su una tecnologia parzialmente trasmittente che permette di guardare contemporaneamente l'immagine virtuale sovrapposta alla vista reale.

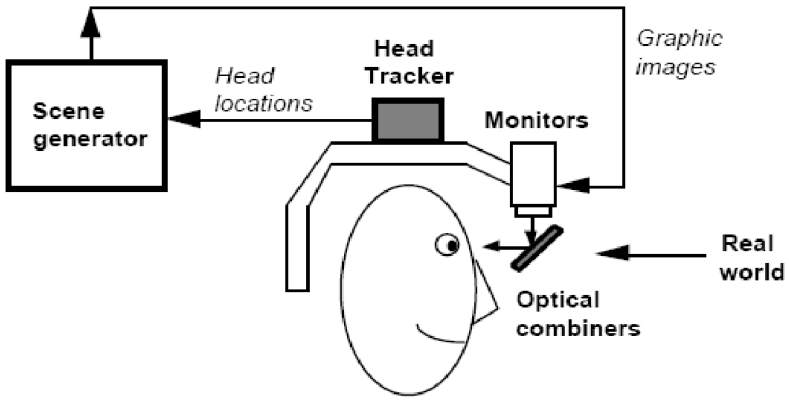

Figura 1.3.2: Schema di un HMD del tipo optical see through
(Tratta da "A survey of Augmented Reality" di Ronald T. Azuma)

Tali visori sono molto simili agli Head Up Display utilizzati dai piloti degli aerei dell'aviazione militare. Una peculiarità di questi sistemi è che riducono l'intensità di luce fino al 30%.

I visori di tipo Video See Through (v. Figura 1.3.3) usano invece due telecamere, una per ciascun occhio, con le quali acquisiscono le immagini del mondo reale, queste vengono inviate per l'elaborazione al video compositor che le proietta sui display, uno per ciascun occhio, arricchite dalle informazioni opportune. La scelta di un

dispositivo di questo genere consente di realizzare effetti grafici più complessi rispetto ai visori di tipo optical see through. Tale tecnologia impone la messa a fuoco della camera su tutta la scena rendendo nitidi alcuni oggetti e sfocate altre parti dell'inquadratura, caratteristica che ne limita il comfort.

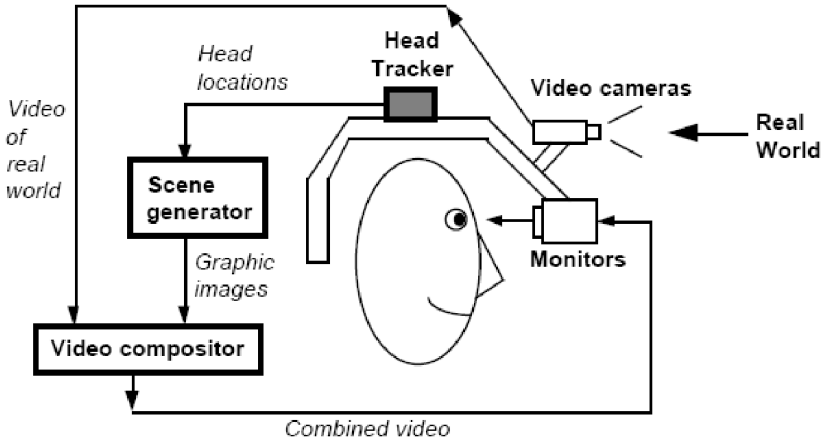

Figura 1.3.3: Schema di un HMD display del tipo video see through (Tratta da "A survey of Augmented Reality" di Ronald T. Azuma)

Gli *hand held display* sono dispositivi portatili quali palmari (PDA), cellulari, smartphone, display a specchio e video proiettori. Tali dispositivi sebbene efficaci dal punto di vista della portabilità presentano alcuni limiti: essendo dei dispositivi che si tengono in mano il tracciamento viene fatto rispetto al dispositivo e non rispetto a dove realmente l'utente sta guardando e le piccole dimensioni del display permettono di variare poco l'angolo di vista.

A differenza dei dispositivi appena visti, i display di tipo spaziale non richiedono all'utente di indossare alcuna strumentazione. L'apparecchiatura è in questo caso installata direttamente nell'ambiente oggetto dell'osservazione. Questa tecnologia di visualizzazione prevede tre tipi differenti di visualizzatori:

- dispositivi *Video See Through*;
- dispositivi *Optical See Through*;
- dispositivi diretti.

I dispositivi di tipo see through utilizzano una videocamera per l'acquisizione e una configurazione PC di tipo desktop; le immagini aumentate vengono proiettate direttamente sul monitor del PC. Tali dispositivi utilizzano combinatori ottici posizionati nell'ambiente, in grado di generare immagini allineate con il punto di

vista dell'osservatore. Esistono tipologie differenti di combinatori ottici: specchi, schermi trasparenti e ologrammi. Gli ologrammi registrano una scena captando ampiezza, lunghezza d'onda e fase. Questo consente di ricostruire il fronte d'onda visivo e generare uno scenario tridimensionale, osservabile da diversi punti di vista.
Gli svantaggi di questo tipo di visualizzatori sono legate all'impossibilità di creare applicazioni portatili, poiché sono legati allo spazio di implementazione. Inoltre consentono un'interattività limitata.

1.3.3 Applicazioni

Dall'inizio degli anni '90 la possibilità offerta dalla AR di far coesistere informazioni reali e virtuali ha trovato applicazione in diversi settori di ricerca.
Molto spesso accade che nella ricerca, la prototipazione e la sperimentazione di nuove tecnologie inizia nel settore militare e viene poi introdotta successivamente in ambito civile.
L'esercito americano fu il primo a introdurre la realtà aumentata con una tecnologia di visualizzazione denominata *Head Up Display* (v. Figura 1.3.4) che permetteva ai piloti dell'aeronautica di visualizzare i dati di volo senza dover abbassare lo sguardo sui vari strumenti nell'abitacolo avendo tali informazioni proiettate sul casco.

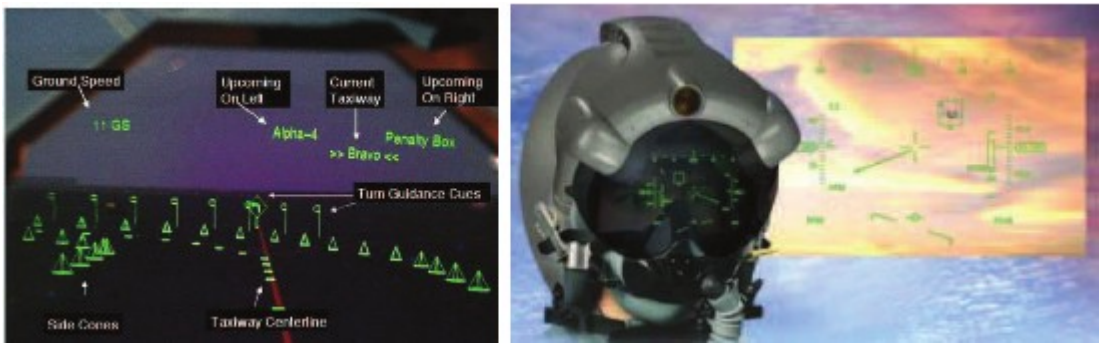

Figura 1.3.4: Visione con un Head Up Display (A sinistra) Head Ud Display (A destra)

Un'altra applicazione di AR in campo militare è la simulazione volta all'addestramento dei soldati. In tale ambito un esempio significativo è il SIMNET, un diffuso sistema di simulazione di giochi di guerra che prevede di equipaggiare il personale militare con caschi provvisti di visori, mediante i quali si proiettano le attività delle altre unità che partecipano all'esercitazione. Lo scenario di guerra può essere aumentato con informazioni supplementari o mettendo in evidenza unità nemiche nascoste. Un esempio riguarda la ricognizione aerea in grado di identificare

oggetti sospetti e trasmetterne la posizione ad unità di terra. Questi oggetti, anche se nascosti alla vista delle forze di terra, vengono visualizzati sui display integrando alle immagini reali dati aggiuntivi.

In campo medico la AR è stata adottata per migliorare l'affidabilità degli interventi chirurgici. In questo settore la ricerca di tecniche chirurgiche poco invasive si scontra con la difficoltà da parte del medico di avere una visione adeguata della zona di intervento. D'altra parte le informazioni provenienti dai sistemi di imaging, quali TAC, ultrasuoni ed MRI (risonanza magnetica), che forniscono informazioni dettagliate sull'anatomia e sulla fisiologia del paziente, possono essere proiettate efficacemente grazie ai sistemi di AR. Un esempio di operazione che utilizza la AR in campo medico è il progetto ARAV (Augmented Reality Aided Verteborplasty), che elaborando le immagini della TAC e della fluoroscopia digitale assiste il medico nell'intervento di riparazione della vertebra fratturata tramite un'iniezione di cemento speciale.

Le interfacce AR permettono di sovrapporre le immagini virtuali al corpo del paziente fornendo una visualizzazione a strati simile ai raggi X per la zona di intervento. In questo modo il medico può eseguire con maggiore precisione procedure complesse quali, la perforazione della scatola cranica per la chirurgia al cervello, biopsie o laparoscopie. Quindi, le applicazioni di AR in campo medico hanno come comune denominatore l'impiego della chirurgia guidata per mezzo di immagini acquisite, attraverso esami clinici, della fisiologia del paziente.

Nella robotica la realtà aumentata è utilizzata per simulare l'effetto di movimentazione automatizzata delle articolazioni meccaniche dei robot (v. Figura 1.3.5).

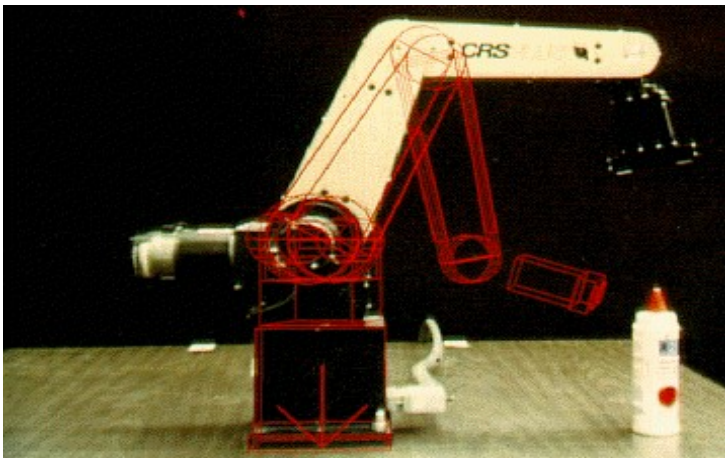

Figura 1.3.5: Simulazione del movimento di un braccio meccanico

Croby e Nafis (1994) descrivono un sistema di *Realtà Aumentata* per la manipolazione di un robot che si occupa delle operazioni di ispezione di un reattore nucleare. In un sistema telerobotico, l'operatore utilizza un'immagine visiva dello spazio di lavoro per guidare il robot. Ad esempio se l'operatore sta per compiere un movimento, questo viene prima simulato su un robot virtuale e visualizzato sul display. A questo punto si può decidere di procedere realmente dopo aver visto i risultati. Il robot quindi esegue direttamente solo i movimenti necessari, eliminando pericolose oscillazioni spesso presenti a causa di ritardi di comunicazione con il sito remoto.

Sin dalla nascita dei primi sistemi, la AR è stata spesso impiegata per assistere nelle operazioni di manutenzione, riparazione ed assemblaggio di macchinari, automobili e aeromobili.

In Figura 1.3.6 si riporta, come esempio, un'applicazione sviluppata di recente dalla casa automobilistica BMW per le operazioni di manutenzione sulle macchine. L'operatore indossa occhiali di tipo Optical See Through integrati da microfono e cuffie. L'applicazione visualizza la sequenza e la tipologia delle operazioni da seguire e l'operatore, attraverso il microfono, comunica all'applicazione l'operazione da visualizzare.

Figura 1.3.6: Applicazione della realtà aumentata per la manutenzione di un'auto

Si registra l'utilizzo della tecnologia della AR in settori di ricerca e attività come quello manifatturiero, logistico, arte e navigazione.

In ambito commerciale le applicazioni di realtà aumentata si concentrano sull'utilizzo di dispositivi mobili, quali smartphone, cellulari e tablet. La maggior parte dei tali applicazioni hanno l'obbiettivo di fornire all'utente informazioni, come indicazioni stradali o dati riguardanti punti di interesse. Un esempio di applicazione di questo genere è Layar (v. Figura 1.3.7), si tratta di un browser che fornisce vari livelli di informazioni rispetto al luogo in cui ci si trova e si sta inquadrando.

Figura 1.3.7: Esempi di indicazioni stradali, informazioni turistiche e informazioni su locali pubblici

Questo tipo di applicazioni necessitano, però, di un sistema di geolocalizzazione e di

una connessione ad Internet per poter scaricare i dati relativi ai luoghi sui quali si vogliono avere informazioni aggiuntive.

Capitolo 2 Tracciamento ottico nei sistemi di realtà Aumentata

Come visto nel capitolo precedente, le applicazioni di AR devono essere in grado di tracciare i vari oggetti del mondo reale, cioè devono individuarli e posizionarli nella scena tridimensionale. In questo capitolo si analizzano matematicamente i sistemi basati sul tracciamento ottico o vision-based di immagini planari e marker.

Queste nozioni vengo introdotte per rendere chiaro il funzionamento del tracciamento ottico utilizzato anche da QCAR SDK.

2.1 Calibrazione della camera

Nei sistemi di realtà aumentata la maggior parte del costo computazionale è relativo all'interpretazione delle immagini, o meglio alla trasformazione delle informazioni provenienti da sistemi di cattura quali telecamere o fotocamere.

La disciplina dell'informatica che si occupa di studiare tale problema è la *Computer Vision*. In questa sezione si introducono i parametri ottici che legano, attraverso trasformazioni geometriche, i punti nello spazio tridimensionale alle loro proiezioni geometriche nello spazio dell'immagine.

2.2 Proiezione prospettica

La creazione di un'immagine, dal punto di vista matematico, consiste nel proiettare i punti dello spazio tridimensionale nello spazio bidimensionale dell'immagine stessa.

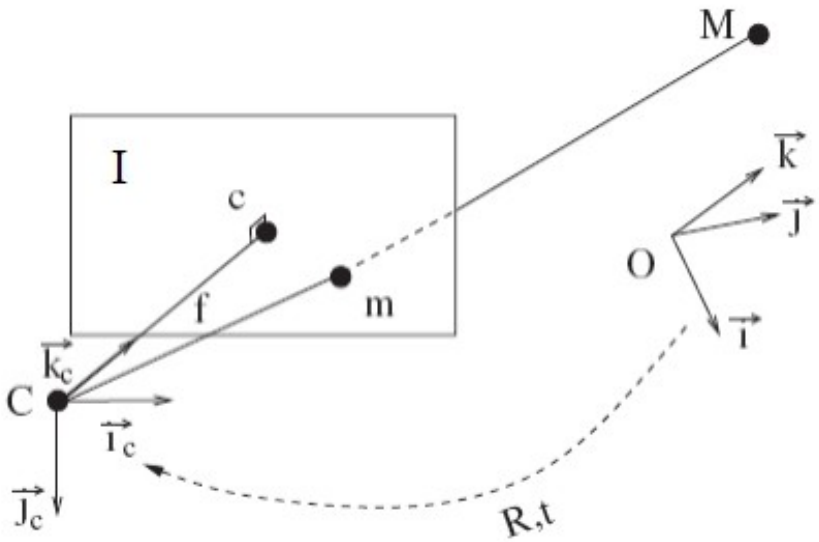

Figura 2.2.1: sistemi di riferimento del modello di proiezione prospettica: camera C, immagine I, spazio reale M.

Facendo riferimento alla Figura 2.2.1, le coordinate in tre dimensioni di un punto *M*, con $M=[X,Y,Z]^T$ nello spazio euclideo, e il punto corrispondente $m=[u,v]^T$ nello spazio immagine sono legati dalla seguente relazione:

$$s \cdot m = P_{3x4} \cdot M \qquad (2.1)$$

dove s è un fattore di scala, mentre *m* e *M* sono le coordinate omogenee dei punti *m* e *M*, infine *P* indica la matrice 3×4 di proiezione prospettica e dipende da 11 parametri. La matrice di proiezione prospettica *P* viene calcolata dal seguente prodotto matriciale:

$$P = K \cdot (R,t) \qquad (2.2)^1$$

La matrice *K* quadrata 3×3 è la matrice di calibrazione della camera, dipende dai parametri intrinseci che collegano le coordinate di un punto dell'immagine *I* con le coordinate corrispondenti nel sistema di riferimento della camera *C*. Mentre la matrice (R,t) è la matrice 3×4 dei parametri estrinseci che definiscono la posizione e orientazione del sistema di riferimento della camera, rispetto al riferimento mondo, che è supposto noto. Più nel dettaglio *R* è la matrice quadrata di ordine tre di rotazione e $t=[t_1\ t_2\ t_3]^T$ è il vettore di traslazione.

La matrice *K* è composta dai parametri intrinseci di calibrazione della camera, nel dettaglio può essere scritta nel seguente modo:

1 La matrice *(R,t)* è una matrice composta dalla matrice *R* a cui viene aggiunto il vettore colonna *t*.

$$K = \begin{bmatrix} \alpha_u & s & u_0 \\ 0 & \alpha_v & v_0 \\ 0 & 0 & 1 \end{bmatrix} \quad (2.3)$$

dove:

- α_u e α_v sono i fattori di scala nelle direzioni u e v dello spazio immagine, proporzionali alla lunghezza focale f della camera: $\alpha_u = k_u \cdot f$ e $\alpha_v = k_v \cdot f$, con k_u e k_v numero di pixel per unità di lunghezza nelle direzioni u e v;
- $c=[u_0\ v_0]$ è il punto che rappresenta l'intersezione dell'asse z della camera con il piano dell'immagine;
- s è il parametro di distorsione ed è diverso da zero solo se gli assi u e v non sono perpendicolari; al giorno d'oggi questa eventualità è molto rara, quindi solitamente è uguale a zero.

In alcuni casi la matrice K assume una forma più semplice: infatti se il numero di pixel per unità di lunghezza nelle direzioni u e v della camera è lo stesso i parametri α_u *e* α_v sono uguali. Un'ulteriore semplificazione è pssibile quando il punto c è perfettamente centrato nello spazio dell'immagine: in tal caso i parametri u_0 e v_0 sono nulli, e si ottiene così la matrice semplificata:

$$K = \begin{bmatrix} \alpha & 0 & 0 \\ 0 & \alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \text{ , con } \alpha = \alpha_u = \alpha_v$$

Quando sono noti tutti i parametri sopraelencati la camera si definisce calibrata.

Assumendo che i parametri interni siano noti attraverso un processo di calibrazione, il problema del tracking si riconduce alla determinazione degli elementi della matrice di posizionamento della camera (R,t). Tale trasformazione consente di esprimere le coordinate di un generico punto P_m dal sistema di riferimento dello spazio euclideo, nel sistema di riferimento della camera (P_c):

$$P_m = R \cdot P_c + t \quad (2.4)$$

Da questa formula si possono così ottenere le coordinate del centro ottico della camera: imponendo $P_c=[0\ 0\ 0]^T$, si ottiene

$$C = -R^{-1} \cdot t = -R^T \cdot t$$

In un sistema AR come QCAR SDK, i sistemi di riferimento sono necessari per risalire alla relazione spaziale che esiste tra l'oggetto e la camera, ossia tra il mondo reale e l'utente.

Il modello appena descritto, per la proiezione dei punti dallo spazio euclideo a quello dell'immagine, risulta incompleto nel caso in cui siano presenti fenomeni di distorsione dell'immagine (v. Figura 2.2.2 a sinistra).

Figura 2.2.2: Immagine distorta (a sinistra) immagine corretta (destra)

La distorsione può essere considerata nella (2.1) introducendo una trasformazione bidimensionale che corregga l'effetto di bombatura sull'immagine dovuto alle imperfezioni ottiche della camera (v. Figura 2.2.2 a destra). Gli effetti distorsivi della camera si possono sintetizzare con due fenomeni, la distorsione radiale e la distorsione tangenziale; molto spesso quest'ultima non viene considerata per il suo contributo infinitesimo.

Definendo $\tilde{U}=[\tilde{u}\,,\tilde{v}]^T$ come le coordinate in pixel dell'immagine distorta e con $\tilde{X}=[\tilde{x}\,,\tilde{y}]^T$ le corrispondenti coordinate normalizzate, si ha:

$$\tilde{u}=u_0+\alpha_u\cdot\tilde{x}$$
$$\tilde{v}=v_0+\alpha_v\cdot\tilde{y}$$

dove u_0, v_0, α_u, α_v sono i parametri interni di calibrazione della camera introdotti con la (2.3). Definendo invece con i $U=[u,v]^T$ e $X=[x,y]^T$ parametri corrispondenti rispettivamente a $\tilde{U}$ e $\tilde{X}$ ottenuti correggendo gli effetti della distorsione, e considerando il contributo della distorsione radiale si ottiene la seguente espressione:

$$\tilde{X}=X+\delta_{radiale}\cdot X$$

dove la distorsione solitamente viene scritta in forma polinomiale ed è definita come segue:

$$\delta_{radiale}=1+k_1\cdot r^2+k_2\cdot r^4+\ldots$$

dove $r=\|X\|=\sqrt{x^2+y^2}$.

2.3 Calcolo della posizione dell'immagine

Una volta calibrata la camera, cioè calcolati i parametri interni, il problema del tracciamento si riconduce al calcolo dei parametri esterni per ogni frame catturato dalla camera in un programma AR. Sostanzialmente si cerca di calcolare il più precisamente possibile i valori dei componenti della matrice *(R,t)* della formula (2.2) attraverso la corrispondenza di punti nello spazio $M_i = [X_i\ Y_i\ Z_i]^T$ con i rispettivi punti individuati nell'immagine $m_i = [u_i\ v_i]^T$

Il problema da un punto di vista matematico si riduce al calcolo degli elementi della matrice *P* attraverso n relazioni del tipo:

$$P \cdot M_i = m_i \tag{2.5}$$

Se, come in questo caso, i parametri intrinseci della camera sono noti, è possibile dimostrare che se le corrispondenze riguardano punti complanari sono sufficienti 4 punti per risolvere il problema, a patto che non vi siano triplette di punti allineati. Il problema, così posto, è denominato *DLT* (Direct Linear Transformation)

Il DLT può essere utilizzato per stimare la matrice *P* anche quando i parametri intrinseci non siano noti. Dalla corrispondenza i-esima tra i punti M_i *e* m_i si ottengono due equazioni indipendenti:

$$\begin{aligned} \frac{P_{11} \cdot X_i + P_{12} \cdot Y_i + P_{13} \cdot Z_i + P_{14}}{P_{31} \cdot X_i + P_{32} \cdot Y_i + P_{33} \cdot Z_i + P_{34}} &= u_i \\ \frac{P_{21} \cdot X_i + P_{22} \cdot Y_i + P_{23} \cdot Z_i + P_{24}}{P_{31} \cdot X_i + P_{32} \cdot Y_i + P_{33} \cdot Z_i + P_{34}} &= v_i \end{aligned} \tag{2.6}$$

Il calcolo dei parametri di *P* dipende fortemente dalla disposizione dei punti processati, con un numero minimo teorico di 6 corrispondenze di punti. Una semplificazione efficace consiste nel separare il calcolo dei parametri intrinseci da quelli estrinseci.

Noti i parametri intrinseci, attraverso la calibrazione, i parametri estrinseci si possono calcolare come segue:

$$K^{-1} \cdot P = (R,t) \tag{2.7}$$

Il DLT consente di calcolare i parametri sia intrinseci che estrinseci della camera, tuttavia nel caso in cui i punti appartengano ad un piano e siano non allineati a gruppi di tre, il calcolo subisce delle semplificazioni.

In questo caso la relazione di proiezione tra i punti del target (v. Figura 2.3.1) e la loro proiezione nello spazio immagine si riconduce al calcolo dei parametri di una matrice 3x3, denominata *matrice omografica*.

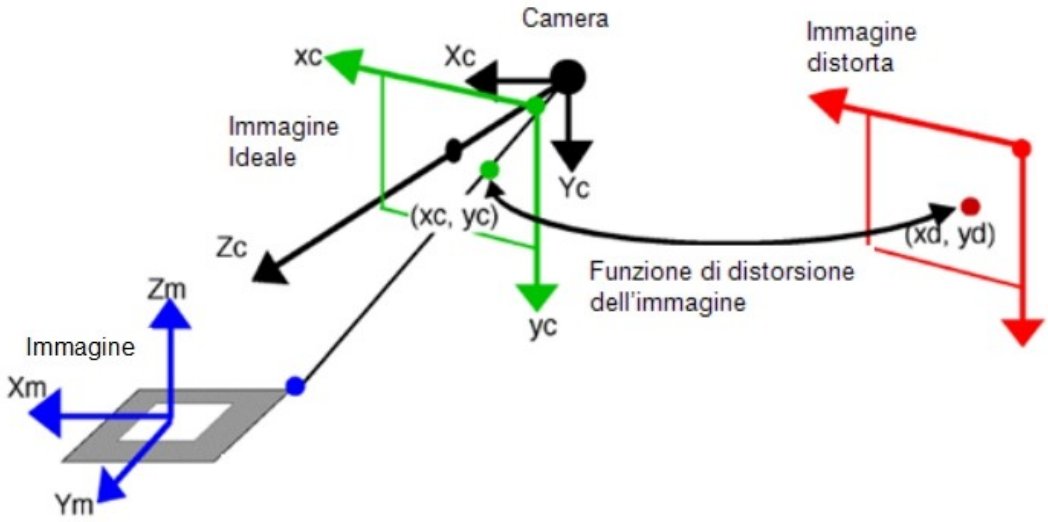

Figura 2.3.1: Calcolo dei parametri di posizionamento

Considerando un sistema di riferimento assegnato al target con il piano dell'immagine coincidente al piano di equazione *Z=0*, la matrice omografica H si ricava a partire dalla matrice (R,t).

Poiché si avranno solo punti del tipo:

$$M_i = \begin{bmatrix} X_i \\ Y_i \\ 0 \\ 1 \end{bmatrix} \tag{2.8}$$

la proiezione di tali punti M_i nello spazio immagine, secondo la (2.1), sarà:

$$\begin{aligned} m &= P \cdot M \\ &= K\left(R_1, R_2, R_3, t\right)\begin{bmatrix} X \\ Y \\ 0 \\ 1 \end{bmatrix} \\ &= K\left(R_1, R_2, t\right)\begin{bmatrix} X \\ Y \\ 1 \end{bmatrix} \\ &= H\begin{bmatrix} X \\ Y \\ 1 \end{bmatrix} \end{aligned} \tag{2.9}[2]$$

Dalla (2.9) si capisce che una volta note le matrici *H* e *K* è possibile calcolare la posizione relativa della camera rispetto all'immagine in esame sfruttando la relazione tra matrice *H* e matrice *P*, nella seguente maniera:

2 La matrice (R_1, R_2, R_3,t) indica una matrice composta dalle colonne 1, 2 e 3 della matrice R e dal vettore colonna t

$$[K]^{-1}[H]=[R_1, R_2, t] \tag{2.10}$$

dove R_1 e R_2 rappresentano i vettori X_m e Y_m del riferimento dell'immagine (v. Figura 2.3.1) espresse nel riferimento della camera, il vettore R_3 si ottiene dal prodotto vettoriale di R_1 e R_2 e il vettore t, che definisce il vettore traslazione tra il punto di vista della camera e l'origine dell'immagine, è noto dalla (2.10). Infine i componenti della matrice H possono essere stimati utilizzando l'algoritmo DLT per almeno quattro punti.

Capitolo 3 SDK di Qualcomm per applicazioni di Realtà Aumentata

Nei capitoli precedenti sono stati introdotti i concetti di realtà aumentata, del suo funzionamento ed è stato analizzato matematicamente il processo di tracciamento. In questa sezione si illustrerà il Software Development Kit (SDK) sviluppato da Qualcomm per la creazione di applicazioni per dispositivi mobili basati su sistemi operativi Android.

3.1 Generalità

Sviluppare applicazioni di AR per dispositivi mobili, come smartphone, tablet o cellulari, richiede una conoscenza approfondita dei componenti hardware del dispositivo e del sistema operativo. Qualcomm ha sviluppato un SDK, chiamato QCAR SDK, che fornisce tutti gli strumenti per rendere più semplice l'utilizzo dello schermo, come visualizzatore della realtà aumentata, e della fotocamera, sia per la cattura dei frame video che come sensore per il tracciamento. Qualcomm rende inoltre disponibile anche uno strumento per la creazione dei possibili target tracciabili.

In figura 3.1.1 vengono evidenziati i due compiti principali dello sviluppatore e come Qualcomm, con i suoi strumenti, rende possibile l'interazione tra l'applicazione e creazione dei target.

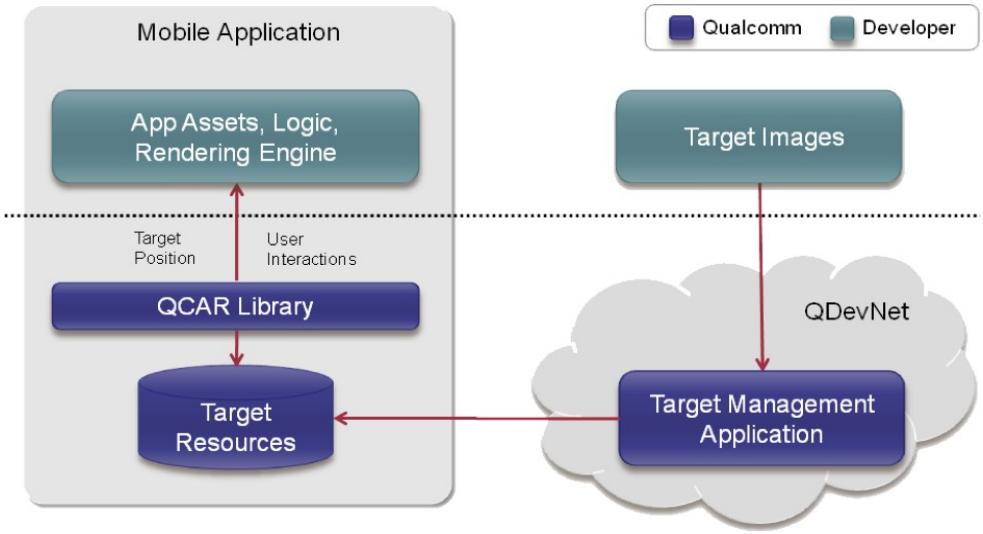

Figura 3.1.1: Fasi di sviluppo di un'applicazione con QCAR SDK (tratto da https://ar.qualcomm.at/qdevnet/developer_guide)

Lo sviluppo dell'applicazione mobile è supportato da QCAR SDK tramite apposite funzioni per il riconoscimento e il tracciamento degli oggetti, che sono contenute in una libreria *shared object*, `libQCAR.so`, distribuita con l'applicazione. Inoltre vengono fornite numerose strutture dati e funzioni specifiche che rendono meno onerosi, a livello computazionale, la lettura e l'interpretazione dei dati degli oggetti tracciati.

Il procedimento di creazione del target, invece, avviene tramite il *Target Management System* (TMS), uno strumento web direttamente accessibile dal sito dedicato alla realtà aumentata di Qualcomm: esso si occupa di calcolare le caratteristiche di una data immagine e salvarle in un formato utile per il confronto con la scena reale durante il tracciamento.

Gli oggetti tracciabili sono principalmente di due tipi: il primo è costituito da una o più immagini planari che posseggono caratteristiche particolari, mentre il secondo è un tipo particolare di marker fiduciale, come un codice QR o un codice a barre matriciale.

Lo sviluppo con QCAR SDK viene svolto in parte nel linguaggio di programmazione Java, utilizzato per creare applicazioni Android, e il rimanente in C++: questa separazione del codice è necessaria per rendere più flessibile l'utilizzo dei vari componenti hardware, oltre che avere maggiore affinità con le librerie preesistenti che sono in codice nativo. Per rendere comunque possibile la comunicazione tra Java e codice nativo si deve utilizzare un framework chiamato JNI (Java Native Interface).

3.2 Architettura QCAR SDK

In figura 3.2.1 vengono illustrati i componenti base che un'applicazione sviluppata con QCAR SDK deve avere:

- camera;
- conversione dell'immagine dal formato proprietario della fotocamera ad un formato utilizzabile per l'analisi dei frame catturati dalla camera;
- tracker: tracciamento di tutti gli oggetti appartenenti ad un frame e gestione degli eventi derivanti dall'analisi delle immagini;
- video background renderer: sincronizzazione tra i frame catturati dalla fotocamera e i frame contenenti l'oggetto virtuale;
- application code;
- target resources: database dei target e delle informazioni delle immagini tracciabili.

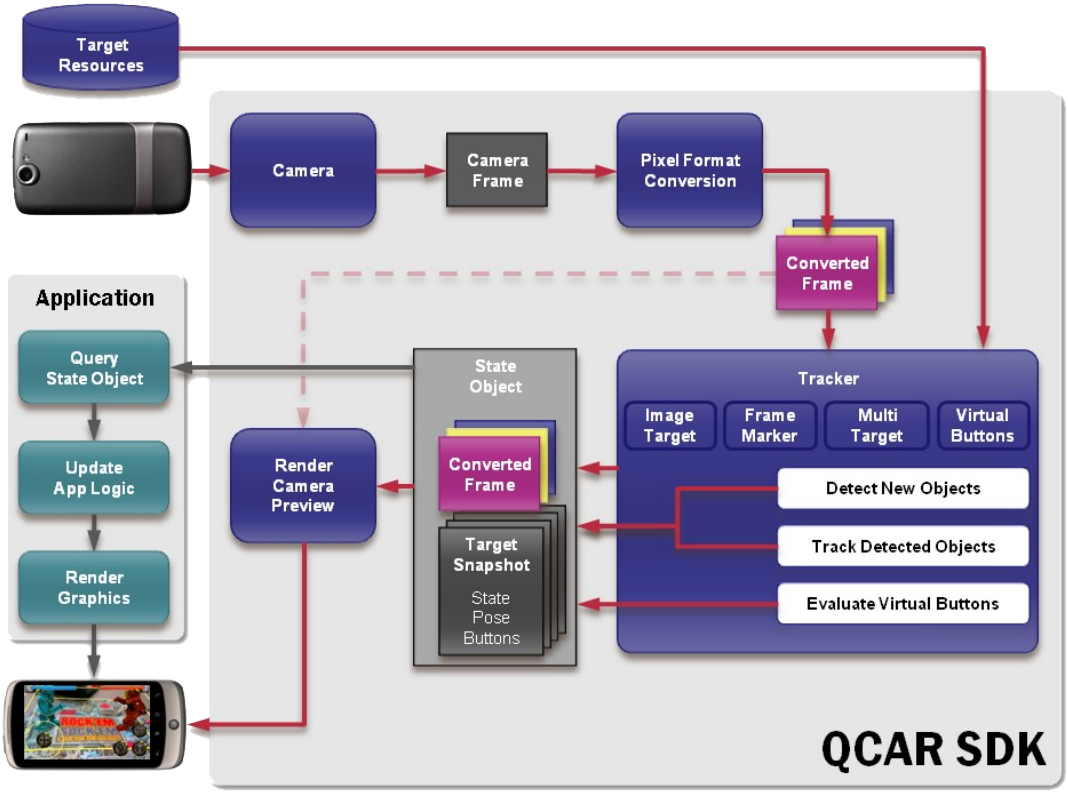

Figura 3.2.1: Schema a blocchi del funzionamento di un'applicazione AR (tratto da https://ar.qualcomm.at/qdevnet/developer_guide)

La creazione di un'unica istanza del blocco *camera* assicura l'efficiente trasferimento del frame catturato al blocco di conversione del formato dell'immagine e

successivamente al blocco per il tracciamento. Lo sviluppatore ha il compito di indicare l'inizio della registrazione delle immagini, quindi la cattura dei frame, e il termine dell'operazione. Ogni frame catturato viene automaticamente consegnato al blocco successivo con il formato e la dimensione dipendenti dal dispositivo ottico. Il frammento di codice 3.2.1 indica le operazioni da seguire per inizializzare la fotocamera e iniziare la cattura del frame.

```
//inizializza la fotocamera se non riesce ad inizializzare esce
if (!QCAR::CameraDevice::getInstance().init())
      return;

//configurazione del formato video per la visualizzazione nello schermo
ConfigureVideoBackground();

//seleziona il modo di cattura della fotocamera se non riesce esce
if (!QCAR::CameraDevice::getInstance().selectVideoMode(
                                  QCAR::CameraDevice::MODE_DEFAULT))
      return;

//inizio cattura
if (!QCAR::CameraDevice::getInstance().start())
      return;
```

Frammento 3.2.1: creazione di un'istanza della fotocamera e inizio cattura dei frame

La conversione dell'immagine avviene tramite il *Pixel Format Converter*, che trasforma il frame dal formato nativo della camera nei formati adatti per le operazioni di rendering, che utilizzano le librerie OpenGL ES, e per il tracciamento. Un esempio di formato compatibile con le librerie grafiche è RGB565[3] mentre un modello basato sulla luminosità è adatto per il tracking. Oltre alla conversione del formato dell'immagine in questo blocco avviene anche una riduzione della risoluzione dell'immagine, che come le due precedenti conversioni viene mantenuta in memoria finché non sarà disponibile un nuovo frame da analizzare.

Anche per il *Tracker* viene creata un'unica istanza. Questo blocco contiene gli algoritmi di computer vision che individuano e tracciano gli oggetti del mondo reale analizzando i frame video precedentemente convertiti in un formato appropriato. Nel frammento di codice 3.2.2 viene avviata l'istanza tracker.

```
QCAR::Tracker::getInstance().start();
```

Frammento 3.2.2: avvio dell'istanza di Tracker

3 È un modello di colore, cioè un modello matematico per rappresentare i colori, per ogni punto; di tipo adattativo basato sui tre colori primari: rosso, verde e blu rispettivamente codificati con 5, 6 e 5 bit

QCAR SDK permette solo il tracciamento di quattro tipologie di target, cioè Image Target, Frame Marker, Multi Target e Virtual Button che verranno descritti successivamente, e ognuna di esse ha algoritmi appropriati per il riconoscimento. Il risultato raggiunto dai diversi algoritmi viene salvato in uno State Object che verrà usato dal renderer e reso accessibile al codice dell'applicazione.

Il blocco *Video Background Renderer* ricrea l'immagine immagazzinata nello State Object nel formato per la visualizzazione del display.

Ogni frame processato può comportare un aggiornamento dello State Object e un conseguente aggiornamento dell'immagine visualizzata nel display, per cui deve essere richiamato il metodo di rendering dall'applicazione. Quindi lo sviluppatore deve procedere seguendo i successivi tre punti:

1. interrogare lo State Object per verificare se sono stati individuati nuovi target, marker o se sono state aggiornate le informazioni relative agli elementi precedentemente individuati;
2. aggiornare le informazioni dell'applicazione relative allo State Object;
3. eseguire il rendering dell'immagine con le informazioni aumentate.

Target Resources: la sorgente del Target viene creata usando uno strumento reso disponibile on-line e chiamato Target Management System. Questo strumento permette di creare i due file che definiscono le immagini tracciabili, uno di configurazione, `config.xml`, che permette al programmatore di configurare con accuratezza le caratteristiche dei trackable e un file binario, `qcar-resources.dat`, che contiene il database dei trackable. Questi due file sono inseriti nel pacchetto di installazione dell'applicazione (apk) al momento della compilazione e vengono usati durante l'esecuzione tramite le librerie fornite dal software development kit.

3.3 Tracciamento con QCAR

Lo sviluppo di un'applicazione tramite QCAR SDK, come visto nelle sezioni precedenti, è principalmente basato sul riconoscimento e sul tracciamento delle immagini; in questa sezione vedremo le classi principali che lo rendono possibile.

3.3.1 Classe Trackable

La classe base Trackable rappresenta tutti gli oggetti del mondo reale che QCAR SDK è in grado di tracciare. Il tracciamento presenta sei gradi di libertà, ovvero le

immagini possono essere riconosciute anche se sono ruotate o traslate (v. Figura 3.3.1).

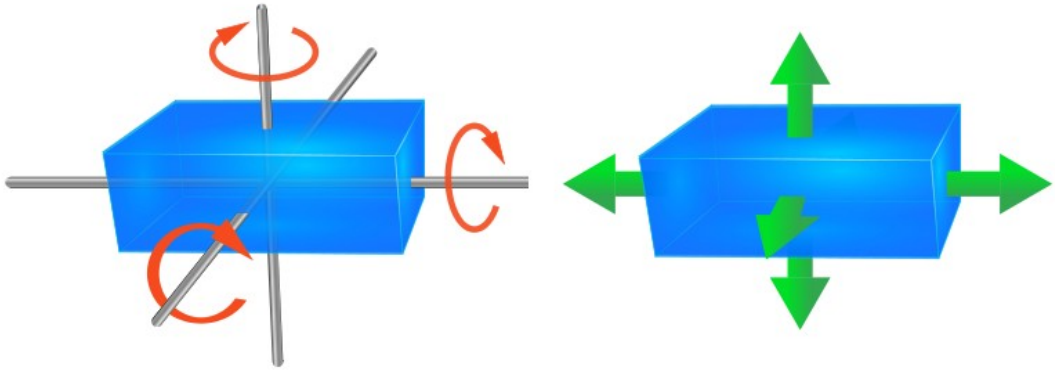

Figura 3.3.1: Movimenti possibili in un sistema con sei gradi di libertà

Ogni trackable, quando riconosciuto e tracciato, possiede delle caratteristiche che lo rendono univocamente riconoscibile: infatti è fornito di un nome, un ID, uno stato e varie informazioni sulla posizione. Image target, Multi target e Marker sono tutte sottoclassi di Trackable che ereditano le proprietà della classe madre (v. Figura 3.3.2), mentre questa è figlia della classe NonCopyable che rende così impossibile la creazione di copie dello stesso oggetto tracciato.

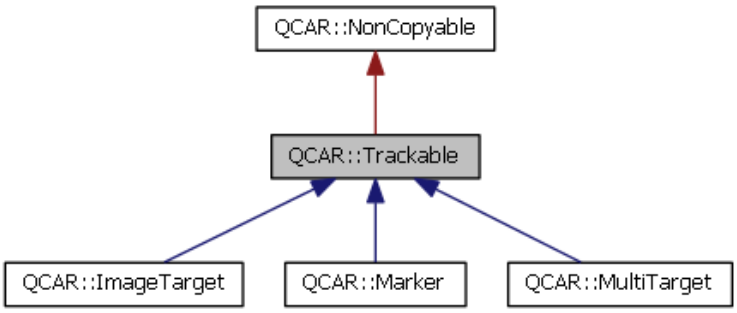

Figura 3.3.2: Ereditarietà della classe Trackable (tratto da https://ar.qualcomm.at/qdevnet/developer_guide)

Gli attributi della classe Trackable vengono aggiornati per ogni frame processato, e il risultato viene trasferito nello State Object per renderlo disponibile all'applicazione.
I parametri che caratterizzano un trackable sono:

- Trackable Type;
- Trackable Name/Identifier;
- Trackable Status.

Ogni oggetto tracciabile può essere di quattro tipi (Trackable Type):

- *UNKNOWN_TYPE*: è un oggetto di tipo trackable di cui non si conosce il tipo;
- *IMAGE_TARGET*: è un trackable di tipo Image Target;
- *MULTI_TARGET*: è un trackable di tipo Multi Target;
- *MARKER*: è un trackable di tipo Marker.

Il riconoscimento del tipo è un punto fondamentale perché fornisce la possibilità di ottimizzare le operazioni successive.

Il nome (Trackable Name) deve individuare univocamente il trackable in esame nel database dei target possibili, deve essere composto al massimo da 64 caratteri.

Ogni oggetto di tipo trackable deve inoltre avere delle informazioni (Trackable Status) associate allo stato in cui si trova nello State Object, che viene aggiornato per ogni frame processato. Lo stato può assumere i seguenti valori:

- *UNKNOWN*: lo stato del trackable è sconosciuto;
- *UNDEFINED*: lo stato del trackable non è definito;
- *NOT_FOUND*: lo stato del trackable non è stato trovato, ad esempio non è contenuto nel database;
- *DETECTED*: il trackable è stato identificato nel frame che si sta analizzando;
- *TRACKED*: il trackable è stato tracciato, cioè se ne conosce la posizione e l'orientamento.

La posizione di un trackable *DETECTED* o *TRACKED*, quindi individuato o tracciato, viene restituita dalla chiamata alla funzione `getPose()` in una matrice 3×4 in row-major order.[4]

La libreria QCAR dispone di semplici funzioni per convertire la matrice ottenuta nella matrice 4×4 di posizionamento della libreria OpenGL ES (v. Frammento di codice 3.3.1), e per proiettare i punti 3D dalla scena nello spazio immagini dello schermo del dispositivo.

```
QCAR::Matrix44F modelViewMatrix =
            QCAR::Tool::convertPose2GLMatrix(trackable->getPose());
```

Frammento 3.3.1: Strumento di conversione tra la matrice di posizione ottenuta dalla camera alla matrice della libreria Opengl ES

4 Una matrice $A=\begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix}$ in row-major order viene salvata linearmente in memoria come segue: 1,2,3,4.

In figura 3.3.3 viene mostrato il sistema di coordinate utilizzato da QCAR SDK nel tracciamento delle immagini. Per gli oggetti che vengono riconosciuti come Image Target o Frame Marker vengono definite delle coordinate locali in cui l'origine degli assi cartesiani, cioè il punto (0,0,0), viene posto al centro dell'immagine o del marker.

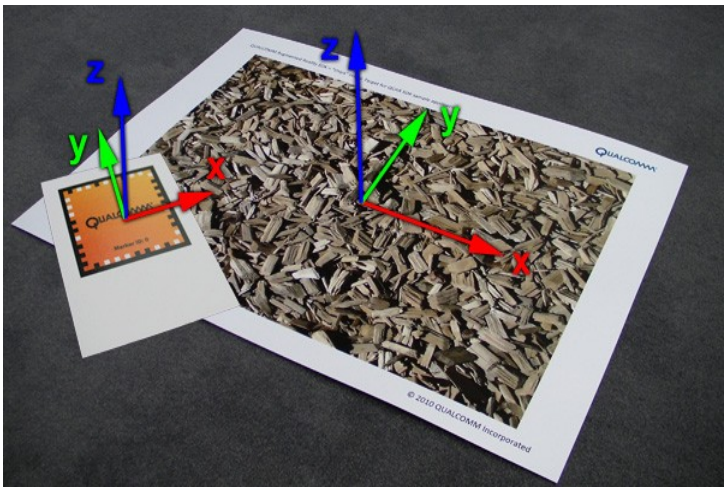

Figura 3.3.3: Sistema di coordinate per i target (tratto da https://ar.qualcomm.at/qdevnet/developer_guide)

L'origine degli assi cartesiani locali per un Multi Target, cioè un insieme di immagini strettamente correlate da una relazione spaziale, viene definita globalmente per l'intera struttura. La posizione dei componenti del Multi Target, cioè le varie Image Target, viene calcolata rispetto al sistema di coordinate globali, indipendentemente dalla prospettiva con la quale è tracciato il Multi Target. Questa caratteristica permette ad un oggetto geometrico di essere tracciato con continuità con le stesse coordinate.

3.3.2 Classe Image Target

Un Image Target, come dice il nome, è un'immagine che QCAR SDK è in grado di individuare e di tracciare. A differenza dei tradizionali marker, data matrix codes e QR codes, un Image Target non necessita di speciali regioni bianche e nere o codici per essere riconosciuto. Mediante l'utilizzo di sofisticati algoritmi QCAR SDK permette di individuare e tracciare l'immagine utilizzandone le caratteristiche naturali, come la luminosità: queste vengono confrontate con le informazioni contenute nel database dei target conosciuti e quindi, in caso il confronto abbia esito positivo, vengono restituite tutte le informazioni che la riguardano. Una volta individuato e riconosciuto un target, QCAR lo traccia fintantoché rimarrà, anche

parzialmente, all'interno dell'inquadratura della camera.

QCAR SDK fornisce la possibilità, di avere fino a cinquanta elementi nel database dei target conosciuti e di individuare e tracciare più di un target per volta fino ad un massimo di cinque contemporaneamente. Le prestazioni in questo caso dipendono molto dalle capacità del processore e della GPU.

Le caratteristiche che personalizzano un Trackable di tipo Image Target sono, oltre a quelle ereditate:

- la dimensione;
- la presenza di pulsanti.

La dimensione ("Target Size"), ottenuta richiamando la funzione `getSize()` che restituisce un vettore contenente altezza e larghezza, è il parametro che indica la misura in millimetri dell'immagine nella scena 3D reale e le informazioni ottenute sulla posizione durante il tracciamento devono essere nella medesima scala. Lo sviluppatore deve specificare questa misura durante la creazione del target con il TMS che creerà il file `config.xml` con tutte le informazioni, oppure può modificare direttamente il file stesso durante lo sviluppo dell'applicazione.

Una caratteristica disponibile per un elemento di tipo Image Taget è la possibilità di avere uno o più pulsanti virtuali (in inglese "Virtual Buttons"). Durante la fase di sviluppo possono essere verificate e modificate le informazioni disponibili nel file `config.xml` riguardanti i pulsanti virtuali relativi ad un dato Image Target ed è possibile valutarne per ognuno lo stato in cui si trova, ovvero se sono attivati o disattivati (v. frammento di codice 3.3.2). I Virtual Buttons possono essere dinamicamente aggiunti o eliminati durante l'esecuzione; essi verranno discussi più approfonditamente nel paragrafo 3.4.

```
<?xml version="1.0" encoding="UTF-8"?>
<QCARConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="qcar_config.xsd">
  <Tracking>
      <ImageTarget size="247 173" name="wood">
      <VirtualButton name="red" rectangle="-108.68 -53.52 -75.75 -65.87"
                                                    enabled="true" />
      <VirtualButton name="blue" rectangle="-45.28 -53.52 -12.35 -65.87"
                                                    enabled="true" />
      <VirtualButton name="yellow" rectangle="14.82 -53.52 47.75 -65.87"
                                                    enabled="true" />
      <VirtualButton name="green" rectangle="76.57 -53.52 109.50 -65.87"
                                                    enabled="true" />
    </ImageTarget>
  </Tracking>
</QCARConfig>
```

Frammento 3.3.2: Contenuto del file `config.xml` *dove sono associati quattro pulsanti virtuali all'immagine "wood"*

3.3.3 Classe Multi Target

Un Multi Target è un oggetto composto da più Image Target che sono strettamente correlati tra loro da una relazione di tipo spaziale. Infatti è sufficiente che venga individuata una porzione dell'oggetto 3D affinché anche le altre parti vengano tracciate essendo conosciuta la relazione che intercorre tra la posizione e l'orientamento della porzione inquadrata e le rimanenti. Finché una frazione del target rimane nell'inquadratura della camera, il Multi Target può essere interamente tracciato.

Un Multi Target non può essere considerato come un semplice insieme di Image Target perché viene considerato da QCAR come un unico oggetto e quindi un unico Trackable con la conseguenza di avere un solo sistema di coordinate cartesiane.

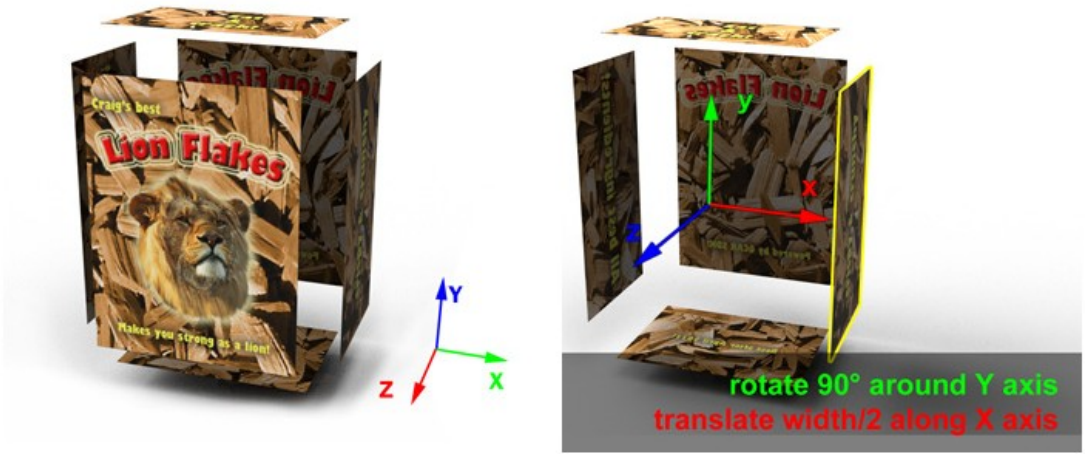

Figura 3.3.4: Sistema di coordinate per un Multi Target (a sinistra), traslazione delle coordinate (a destra) (tratto da https://ar.qualcomm.at/qdevnet/developer_guide)

Le parti che compongono un Multi Target sono create con il TMS da immagini in formato JPG o PNG. Le caratteristiche estratte vengono immagazzinate in un database e usate durante l'esecuzione per il confronto con le immagini catturate dalla fotocamera. Le informazioni sulle dimensioni di ogni Image Target che compone l'oggetto sono contenute nel file `config.xml`; il medesimo file contiene le relazioni spaziali che intercorrono tra i vari elementi del Multi Target, quali traslazione e rotazione (v. Figura 3.3.4).

Oltre al sistema appena descritto la classe MultiTarget mette a disposizione vari strumenti per poter creare un Multi Target aggiungendo o rimuovendo le varie Image Target durante l'esecuzione, con la condizione che la loro relazione spaziale resti invariata, utilizzando le seguenti funzioni:

- `int QCAR::MultiTarget::addPart ( Trackable* trackable);`
  questa funzione aggiunge una parte all'oggetto di tipo Multi Target restituendone l'indice assegnato (v. Frammento di codice 3.3.3);
- `bool QCAR::MultiTarget::removePart ( int idx);`
  questa funzione rimuove la parte che è identificata dall'indice *idx*;
- `bool QCAR::MultiTarget::setPartOffset( int idx, const Matrix34F& offset);`
  questa funzione imposta la traslazione e la rotazione definita dalla matrice *offeset* del trackable di indice *idx* e restituisce *false* se l'indice non è valido o se il tracker è attivo.

```
int numAdded = 0;
for(int i=0; i<6; i++)
{
    if(QCAR::ImageTarget* it = findImageTarget(names[i]))
    {
        int idx = mit->addPart(it);
        QCAR::Vec3F t(trans+i*3),a(rots+i*4);
        QCAR::Matrix34F mat;
        QCAR::Tool::setTranslation(mat, t);
        QCAR::Tool::setRotation(mat, a, rots[i*4+3]);
        mit->setPartOffset(idx, mat);
        numAdded++;
    }
}
```

Frammento 3.3.3: creazione di un Multi Target da sei Image Target distinte

```
struct MyUpdateCallBack : public QCAR::UpdateCallback
{
    virtual void QCAR_onUpdate(QCAR::State& state)
    {
        if(mit!=NULL)
        {
            mit->removePart(5);
        }
    }
} myUpdateCallBack;
```

Frammento 3.3.4: rimozione di una parte del Multi Target con il tracker attivo

È importante notare che l'aggiornamento di un Multi Target comporta un cambiamento interno al tracker stesso, quindi tali cambiamenti non possono essere eseguiti se il tracker è attivo: l'unico modo per eseguire le riconfigurazioni consiste nel fermare il tracker. Comunque queste operazioni sono estremamente costose dal punto di vista computazionale.

L'unico modo per modificare la realazione spaziale che intercorre tra le varie Image Target consiste nella modifica del file `config.xml` che dovrà essere poi reinserito durante la compilazione dell'applicazione.

Le caratteristiche che contraddistinguono ogni componente di un Multi Target sono: il nome, la traslazione e la rotazione rispetto al sistema di coordinate.

3.3.4 Classe Frame Marker

Come visto nelle sezioni precedenti QCAR SDK è in grado di tracciare immagini planari che abbiano delle caratteristiche particolari, come un sufficiente contrasto e una certa ricchezza di dettagli, che le rendono facilmente riconoscibili dal software. Oltre a queste immagini allo sviluppatore viene data la possibilità di tracciare speciali marker fiduciali, chiamati Frame Marker. Questi non sono altro che immagini con caratteristiche particolari, infatti ogni immagine di questo tipo è codificata con un modello binario e deve avere una cornice nera con dei quadratini neri o bianchi disposti sul bordo interno (v. Figura 3.3.6).

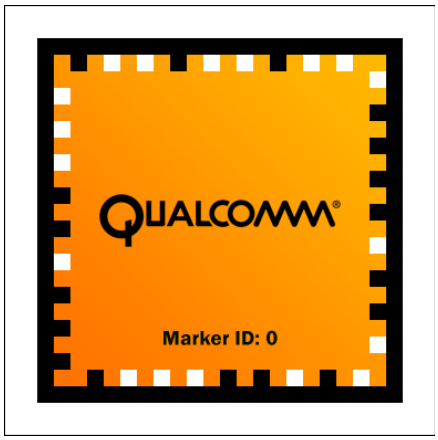

Figura 3.3.5: Esempio di un Frame Marker (tratto da https://ar.qualcomm.at/qdevnet/developer_guide)

Ogni marker è univocamente riconosciuto da un identificativo che è espresso dal modello binario intrinseco alla cornice. Una particolarità di questo tipo di marker è la possibilità di inserirne all'interno una qualsiasi immagine, ad esempio il logo di un'azienda, purché mantenga il contrasto con la cornice. Questa particolarità rende il marker più gradevole alla vista e facilmente riconoscibile, aumentandone l'interesse commerciale.

I parametri che definiscono un frame marker sono i seguenti.

- Dimensione del marker: indica la dimensione attuale del marker nell'unità di misura della scena reale. La dimensione deve comunque essere specificata dallo sviluppatore nel file di configurazione oppure durante l'esecuzione: ciò serve per poter scalare correttamente l'immagine durante il tracciamento.
- Identificativo (ID): ogni Frame Marker codifica il suo ID nel modello binario intorno alla cornice. Questo ID è un numero compreso tra 0 e 511 ottenendo così 512 possibili identificativi.
- Tipo: il tipo di marker può essere solamente di due tipi:
    - *INVALID*: il marker è riconosciuto ma non può essere identificato;
    - *ID_FRAME*: nel marker è codificato correttamente il suo ID.

Il tracciamento dei Frame Marker da parte di QCAR SDK, a differenza di quanto avviene per Image Target e Multi Target, necessita che la cornice e il modello binario siano interamente visibili nell'inquadratura in modo da essere riconosciuti univocamente senza creare ambiguità tra due target di questo tipo simili.

La facilità di decodifica, dal punto di vista computazionale, dei Frame Marker ha dato la possibilità di poter creare applicazioni che li contengono tutti, anche se ne resta limitata la tracciabilità ad un numero massimo di cinque contemporaneamente (vedi frammento di codice 3.3.4).

```
for(int tIdx = 0; tIdx < state.getNumActiveTrackables(); tIdx++)
{
    // Get the trackable:
    const QCAR::Trackable* trackable = state.getActiveTrackable(tIdx);
    QCAR::Matrix44F modelViewMatrix =
        QCAR::Tool::convertPose2GLMatrix(trackable->getPose());

    // Choose the texture based on the target name:
    int textureIndex = 0;

    // Check the type of the trackable:
    assert(trackable->getType() == QCAR::Trackable::MARKER);
    const QCAR::Marker* marker = static_cast<const
                                        QCAR::Marker*>(trackable);
    textureIndex = marker->getMarkerId();

    assert(textureIndex < textureCount);
    const Texture* const thisTexture = textures[textureIndex];
}
```

Frammento 3.3.5: Tracciamento dei Frame Marker

Dal punto di vista del rendering i marker non differiscono dagli Image Target per cui nel rendering si può sovrapporre ad essi qualsiasi oggetto virtuale.

3.4 Virtual Buttons

Gli oggetti di tipo Virtual Button sono delle regioni rettangolari di un Image Target definite in fase di sviluppo, che quando vengono premute o oscurate alla vista della fotocamera generano un evento (v. Figura 3.4.1).

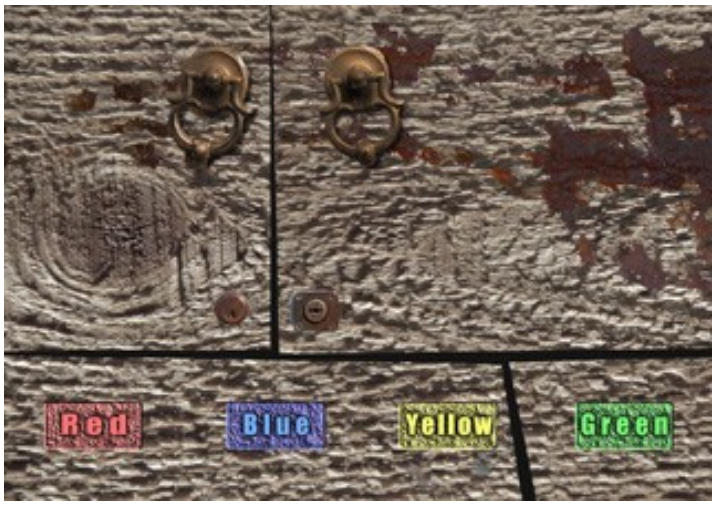

Figura 3.4.1: Image Target con Virtual Buttons (tratto da https://ar.qualcomm.at/qdevnet/developer_guide)

Questi pulsanti virtuali possono essere utilizzati per implementare eventi quali la reale pressione di un pulsante oppure come sensori per rilevare se quella specifica area dell'immagine è coperta da un oggetto. I pulsanti virtuali vengono valutati solamente se sono all'interno dell'inquadratura e se la camera è sufficientemente stabile, ovvero non compie movimenti rapidi; se la fotocamera non è ferma QCAR SDK rimuove la possibilità di valutare l'esistenza dei Virtual Button.

I pulsanti virtuali possono essere definiti direttamente nel file di configurazione `config.xml` come proprietà dell'immagine, oppure possono essere aggiunti o eliminati durante l'esecuzione tramite un insieme apposito di funzioni definito nelle API; è da notare che per apportare tali modifiche il tracker deve essere non attivo (v. frammento di codice 3.4.1).

```
bool toggleVirtualButton(QCAR::ImageTarget* imageTarget, const char* name,
     float left, float top, float right, float bottom)
{
    bool buttonToggleSuccess = false;

    QCAR::VirtualButton* virtualButton = imageTarget
                                            ->getVirtualButton(name);
    if (virtualButton != NULL)
    {
        // Destroying Virtual Button
        buttonToggleSuccess = imageTarget
                                ->destroyVirtualButton(virtualButton);
    }
    else
    {
        // Creating Virtual Button
        QCAR::Rectangle vbRectangle(left, top, right, bottom);
        QCAR::VirtualButton* virtualButton = imageTarget
                                ->createVirtualButton(name, vbRectangle);
        if (virtualButton != NULL)
            buttonToggleSuccess = true;
    }
    return buttonToggleSuccess;
}
```

Frammento 3.4.1: Aggiunta o distruzione di un Virtual Button

Ogni elemento di tipo Virtual Button è caratterizzato dalle seguenti proprietà.

- Nome: come ogni elemento di tipo Image Target o Multi Target anche i pulsanti virtuali vengono identificati univocamente all'interno del target tramite un nome. Questo è composto al massimo da 64 caratteri presi dall'insieme delle lettere maiuscole e minuscole dalla *a* alla *z* dai numeri da *0* a *9* e dai simboli dell'insieme [-, _, .].
- Coordinate del pulsante: i pulsanti sono delle aree rettangolari all'interno dell'Image Target; lo sviluppatore dell'applicazione deve specificare le coordinate dell'angolo in alto a sinistra e dell'angolo in basso a destra del rettangolo, tenendo conto che il centro delle coordinate del sistema locale è il centro dell'immagine.
- Sensibilità del pulsante: questo parametro definisce la velocità di risposta del pulsante e può essere di tre gradi diversi:
    - HIGH: l'occlusione del pulsante viene rilevata immediatamente generando l'evento desiderato. Questa alta sensibilità rende il pulsante molto reattivo ma può provocare anche falsi positivi;
    - MEDIUM: è un compromesso tra HIGH e LOW;

- LOW: il pulsante deve rimanere premuto a lungo perché venga generato l'evento collegato.

3.5 Target Management System

Oltre a QCAR SDK cioè il software per la creazione delle applicazioni, come accennato precedentemente, Qualcomm mette a disposizione il Target Management System (TMS): uno strumento web per rendere facile la creazione dei target su cui poi si andrà a sviluppare il programma. Il TMS è uno strumento pratico e semplice da usare per creare questo insieme di target che poi verrà inserito e distribuito con l'applicazione.

Un target è il risultato dell'elaborazione delle caratteristiche naturali dell'elemento da tracciare che vengono usate dall'applicazione durante l'esecuzione per riconoscere correttamente l'oggetto in esame nella scena reale.

Il TMS fornisce la possibilità di creare vari progetti, che conterranno l'insieme delle immagini usate per creare il database dei target sorgente da scaricare. L'applicazione durante l'esecuzione accetta solamente un database di target sorgente, con la possibilità di contenere più immagini target che saranno utilizzate per il tracciamento e il riconoscimento.

Solitamente per una nuova applicazione si crea un nuovo progetto dove si possono caricare le immagini destinate a comporre l'insieme di target, fatto ciò il TMS consente di scaricare un file compresso contenente il database dei target e l'insieme delle caratteristiche delle immagini che compongono il database.

3.5.1 Creazione di un ImageTarget

Il TMS fornisce la possibilità di gestire tre diversi tipi di target: immagini piane, parallelepipedi e cubi. In questa sezione vengono descritte le modalità di creazione di un target piano e le caratteristiche che deve possedere per essere tracciato.

Per prima cosa con questo sistema si deve scegliere di creare un nuovo target, successivamente viene chiesto di inserire il nome e il tipo di target che si vuole creare ed infine le sue dimensioni espresse in millimetri (v. figura 3.5.1)

Figura 3.5.1: Creazione di un nuovo trackable Image Target (tratto da https://ar.qualcomm.at/qdevnet/developer_guide)

Il passo successivo consiste nel caricare l'immagine, che deve essere ricca di dettagli e non avere porzioni ripetute, come la foto di una facciata di un edificio moderno, oltre ad avere un buon contrasto per renderne più facile il riconoscimento e quindi il tracciamento. Una volta caricata l'immagine il sistema calcolerà e visualizzerà le varie caratteristiche oltre a stimare la qualità per il tracciamento tramite un indicatore da una a cinque stelle (v. figura 3.5.2).

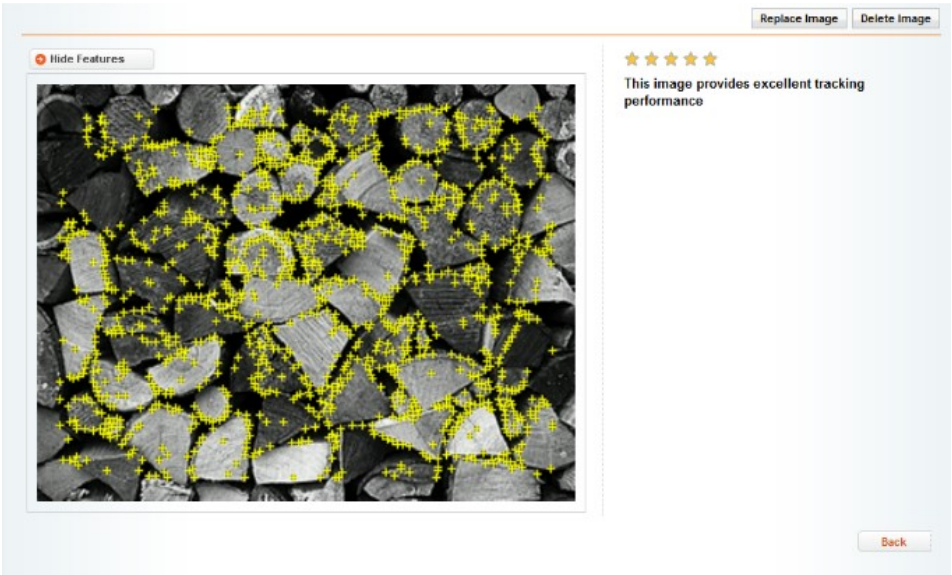

Figura 3.5.2: Elaborazione caratteristiche di un'immagine

3.5.2 Creazione di un MultiTarget

La creazione di un Multi Target è molto simile alla precedente, solamente si deve

selezionare la forma del cubo o del parallelepipedo, ed inoltre verranno richieste le dimensioni dell'oggetto, quindi altezza, larghezza e profondità. Successivamente si caricheranno le varie immagini che andranno a comporre le facce del solido (v. Figura 3.5.3).

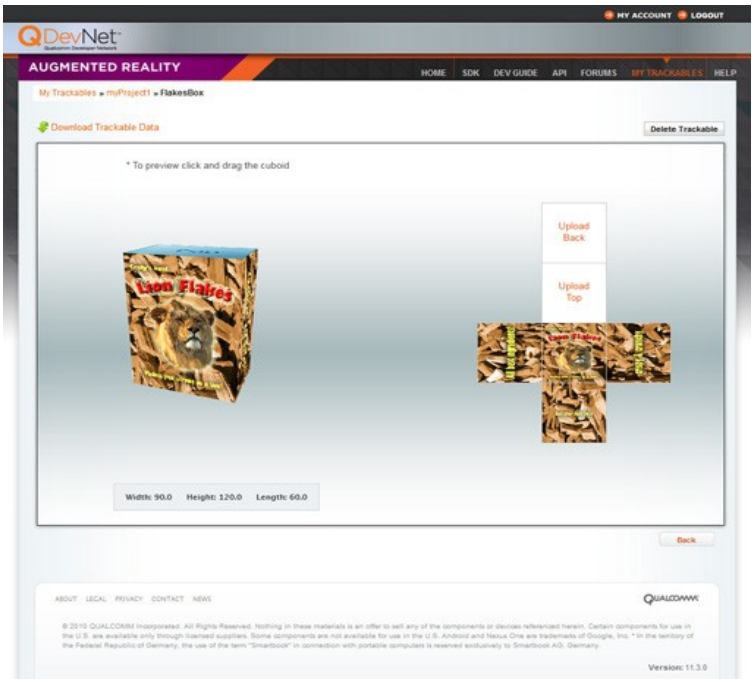

Figura 3.5.3: Creazione di un Multi Target (tratto da https://ar.qualcomm.at/qdevnet/developer_guide)

Il sistema calcolerà automaticamente, oltre alla caratteristiche delle immagini che compongono il Multi Target, anche la relazione spaziale che intercorre tra i vari elementi.

3.5.3 Download e utilizzo dei target

Il TMS, una volta eseguite le operazioni di creazione di un progetto e dei target, permette di scaricare i dati necessari da utilizzare all'interno dell'applicazione. Questi vengono forniti in due file: `config.xml` contenente la configurazione dei vari target, e `qcar-resouces.dat` contenente il database delle caratteristiche delle varie immagini.

Durante l'esecuzione dell'applicazione questi due file dovranno essere caricati per renderne possibile l'accesso, quindi all'interno del codice di inizializzazione dell'applicazione dovranno essere presenti anche le seguenti istruzioni, da eseguire una ed una sola volta:

```
case APPSTATUS_INIT_TRACKER:

      //caricamento del data set di informazioni
    try
    {
        loadTrackerTask = new LoadTrackerTask();
        loadTrackerTask.execute();
    }
    catch (Exception e)
    {
        Tools.LOGE("Loading tracking data set failed");
    }
    break;
```

Frammento 3.5.1: caricamento dei target

Capitolo 4 Creazione di un'applicazione con QCAR SDK

In questo capitolo verrà descritta la creazione una semplice applicazione di realtà aumentata utilizzando gli strumenti resi disponibili da QCAR SDK descritti nel capitolo precedente.

4.1 Strumenti utilizzati

Lo sviluppo di un'applicazione di AR con QCAR SDK per dispositivi con sistema operativo Android richiede l'utilizzo dei seguenti strumenti.

- Java SE Development Kit 6 (JDK): è il linguaggio di programmazione usato per lo sviluppo di applicazioni in Android.
- Eclipse IDE: si tratta di un interfaccia grafica per facilitare la fase di sviluppo, supporta molto linguaggi di programmazione e si integra perfettamente con tutti gli strumenti per lo sviluppo di applicazioni destinate al sistema operativo Android.
- Android SDK: è il sistema di sviluppo per la creazione delle applicazioni; allo sviluppatore fornisce molti strumenti per rendere semplice l'utilizzo dei componenti hardware oltre a fornire un emulatore di dispositivi Android nel quale poter provare le applicazioni durante lo sviluppo.
- Android Development Tool (ADT): è un'estensione di Eclipse IDE per interfacciarsi con Android SDK e quindi facilitarne l'utilizzo.
- Android NDK: è l'ambiente di sviluppo per il codice nativo di Android, ovvero codice ottenuto compilando sorgenti scritti in C o C++.
- Cygwin: è uno strumento che permette la compilazione del codice nativo Android in un ambiente di sviluppo Microsoft Windows.

- QCAR SDK: è il sistema di sviluppo per applicazioni di realtà aumentata per Android. Questo strumento, come descritto nel capitolo precedente, fornisce gli strumenti per il tracciamento di un oggetto; per l'esempio che si illustrerà viene utilizzata la versione 1.0.6.
- Dispositivo mobile: le applicazioni sviluppate con QCAR SDK non possono essere testate con l'emulatore di dispositivi fornito con Android SDK, quindi risulta necessario disporre di un dispositivo fisico nel quale siano presenti almeno il sistema operativo Android versione 2.2, una fotocamera e, ovviamente, uno schermo. Per le prove è stato utilizzato il dispositivo *ARCHOS 32 internet tablet*.

4.2 Utilizzo della Java Native Interface (JNI)

Nelle sezioni precedenti è stata evidenziata la necessità di sviluppare l'applicazione sia in codice Java che in codice nativo. Per rendere possibile l'integrazione del primo con il secondo e viceversa, Java dispone della Java Native Interface (JNI).

La JNI è un'interfaccia che permette di richiamare all'interno del codice scritto in linguaggio di programmazione Java funzioni scritte in codice nativo, o viceversa. Nel nostro caso questa interfaccia è usata per richiamare funzioni implementate in C++ nel programma scritto in linguaggio java che è mormalmente utilizzato per lo sviluppo di applicazioni Android.

4.2.1 Dichiarazione Java di una funzione in codice nativo

La dichiarazione del metodo nella classe Java viene effettuata utilizzando la parola chiave *native*, ad esempio:

```
public native void nativeFunction();
```

L'utilizzo della funzione dichiarata nella classe risulta possibile solamente se viene caricata la libreria nella quale la funzione è implementata. Questo viene fatto tramite le seguenti linee di codice:

```
static
{
	System.loadLibray(nome_della_libreria);
}
```

dove *nome_della_libreria* è un oggetto di tipo *String* che indica, appunto, il nome della libreria nella quale si trova la funzione nativa, queste linee di codice verranno eseguite automaticamente all'avvio del programma; per fare ciò la libreria dovrà

essere caricata nella classe che estende *Activity*: classe che avvia le applicazioni di Android.

4.2.2 Implementazione della funzione in C++

Parallelamente ai metodi dichiarati nella classe Java, in codice nativo si implementano le funzioni utilizzando la seguente dicitura.

```
#include <jni.h>

extern “C”
{
	JNIEXPORT void JNICALL
	Java_nome_package_Nome_Classe_nativeFunction(JNIEnv *, jobject)
	{
		//corpo del metodo
	}
}
```

Frammento 4.2.1: Implementazione di una funzione in codice nativo

Come si vede dal frammento di codice per poter implementare le funzioni native sarà necessario includere anche un *header file* particolare chiamato *jni.h*, che contiene tutte le informazioni che rendono il codice C++ utilizzabile in Java o viceversa, comprese le dualità tra i tipi di dato tra i due diversi linguaggi di programmazione; l'intestazione deve seguire la struttura indicata nel frammento di codice 4.2.1, anche se possono esserci alcune differenze come un numero maggiore di attributi oppure la parola chiave *void* potrebbe essere sostituita dal tipo dell'oggetto che varrà restituito.
Terminata l'implementazione di tutto il codice nativo dovrà essere compilato generando una libreria *shared object*.

4.3 Sviluppo dell'applicazione

In questa sezione vedremo come sviluppare un'applicazione di realtà aumentata basata su QCAR SDK per Android. Sarà necessario disporre dei file generati con il TMS per il tracciamento e il file con la struttura dell'oggetto virtuale.
La creazione di un'applicazione, come detto più volte, avviene su due fronti, lo sviluppo del codice Java e lo sviluppo del codice nativo in C++ e si vedranno quali parti del codice saranno sviluppate nell'uno e quali nell'altro.

4.3.1 Codice Java

Un'applicazione Android ha la necessità di una classe che estenda la classe Activity e implementi i seguenti metodi.

- `protected void onCreate(Bundle savedInstanceState)`: rende possibile la

creazione dell'applicazione e quindi la sua esecuzione.

- `protected void onResume()`: rende possibile il ripristino di un'applicazione dopo essere stato interrotta da un altro processo.
- `protected void onPause()`:rende possibile la pausa di un'applicazione per poi essere ripristinata con la chiamata al metodo *onResume()*.
- `protected void onDestroy()`: chiamata prima che l'applicazione venga distrutta liberando la memoria.

Altresì sarà necessario creare almeno altre tre classi ognuna con uno scopo diverso, anche se due saranno strettamente correlate: infatti una estenderà la classe `GLSurfaceView` di Android che ha il compito di creare e gestire la memoria e la superficie per visualizzare il rendering, mentre l'altra implementerà l'interfaccia `GLSurfaceView.Renderer` che eseguirà in rendering vero e proprio una volta che verrà creata un'istanza della classe `GLSurfaceView`. L'ultima classe necessaria è di supporto per il caricamento e la gestione della struttura dell'oggetto virtuale che dovrà essere costruito in associazione all'immagine tracciata.

Molte parti di codice o funzioni intere devono essere implementate in codice nativo poiché si tratta di codice critico che richiede elevate prestazioni, perché si combina con framework di livello inferiore tipo l'accesso alla camera oppure le librerie OpenGL ES. Le funzioni sviluppate in codice nativo sono poi richiamate nel codice Java tramite apposite definizioni di metodi come spiegato nella sezione 4.2.

In conclusione in codice Java viene sviluppato il codice per la creazione dell'applicazione, la creazione e gestione della superficie di visualizzazione e il caricamento dei file necessari per il tracciamento e per la creazione dell'oggetto virtuale; il resto dell'applicazione viene sviluppato in C++

4.3.2 Codice C++

Le principali funzioni sviluppate in codice nativo, come spiegato precedentemente, sono quelle che si devono interfacciare con i framework di livello inferiore e con le librerie OpenGL ES. Tali funzioni implementano:

- inizializzazione dell'applicazione;
- inizializzazione del rendering;
- l'inizio della cattura dei frame della camera;
- il termine della cattura dei frame della camera;

- il rendering mediante l'utilizzo delle librerie grafiche.

In codice nativo si possono comunque sviluppare tutte le funzioni di supporto che sono necessarie per una corretta esecuzione del codice.

4.4 Esempio di un'applicazione

In questa sezione verrà analizzato un esempio fornito direttamente da Qualcomm al quale sono state apportate alcune modifiche per capirne il funzionamento. L'esempio preso in esame è una semplice applicazione di tracciamento di un'immagine alla quale vine sovraimposta una teiera il cui colore è abbinato all'immagine stessa. Il progetto che implementa tale applicazione si chiama ImageTarget e viene installato assieme a QCAR SDK.

4.4.1 Codice Java dell'esempio

Il codice Java contenuto nel progetto ImageTarget è organizzato in cinque file:

- *DebugLog.java*;
- *ImageTarget.java*;
- *ImageTargetRenderer.java*;
- *QCARSampleGLView.java*;
- *Texture.java*.

Il file *DebugLog.java* è costituito da una classe di supporto che aiuta nelle fasi di sviluppo ad ottenere la cronologia degli eventi (in inglese log) e quindi, in caso si manifestassero errori o malfunzionamenti, ne viene facilitata l'analisi.

Il file *ImageTarget.java* contiene tre classi.

- `ImageTarget`: che estende la classe `Actitity` contenuta nelle SDK di Android.
- `InitQCARTask`: estende la classe `AsyncTask` ed è privata, quindi visibile solo dalla classe `ImageTarget`, ha il compito di inizializzare QCAR in modo asincrono sovrascrivendo il metodo della classe madre

  ```
  protected Boolean doInBackground(Void... params)
  ```

  e alla fine dell'inizializzazione restituisce lo stato del processo utilizzando il metodo sovrascritto

  ```
  protected void onPostExecute(Boolean result)
  ```

  se il parametro *result* è impostato a false si è verificato un errore; in tal caso viene contestualmente generato un evento informando l'utilizzatore dell'impossibilità di inizializzare QCAR.

- `LoadTrackerTask`: anche questa classe come la precedente estende la classe `AsyncTask` e utilizzando i metodi descritti sopra carica i file necessari per inizializzare il tracker, in caso di errore restituisce un messaggio opportuno.

La classe `ImageTarget`, oltre ai metodi principali che sovrascrivono quelli ereditati dalla classe `Activity`, possiede un altro metodo fondamentale per l'esecuzione dell'applicazione, che deve essere richiamato nel metodo `onCreate(Bundle savedInstanceState)` e si occupa dell'inizializzazione di tutti gli elementi e avvia il rendering dell'oggetto (vedi frammenti di codice 4.4.1, 4.4.2, 4.4.3).

Il metodo `updateApplicationStatus(int appStatus)` è sincronizzato, ovvero può essere eseguito solamente da un processo alla volta per evitare che informazioni condivise da più processi vengano modificate più volte contemporaneamente compromettendo l'esecuzione dell'applicazione.

```
private synchronized void updateApplicationStatus(int appStatus)
{
   // esci se non è stato effettuato un cambiamento di stato
   if (mAppStatus == appStatus)
       return;

   // salva il nuovo stato
   mAppStatus = appStatus;

   // esegui azioni specifiche per lo stato in cui si trova l'applicazione
   switch (mAppStatus)
   {
       case APPSTATUS_INIT_APP:
              // inizializza gli elementi non correlati con QCAR
              initApplication();

              // procedi con l'inizializzazione di QCAR
              updateApplicationStatus(APPSTATUS_INIT_QCAR);
              break;

       case APPSTATUS_INIT_QCAR:
              // inizializzazione di QCAR che verrà eseguita una sola volta
              try
              {
                     mInitQCARTask = new InitQCARTask();
                     // se execute va a buon fine procede direttamente con
                     //inizializzazione dell'AR
                     mInitQCARTask.execute();
              }
              catch (Exception e)
              {
                     DebugLog.LOGE("Initializing QCAR SDK failed");
              }
              break;

       case APPSTATUS_INIT_APP_AR:
              // inizializza elementi specifici per l'applicazione di AR
              initApplicationAR();

              // procedi con l'inizializzazione del tracker
              updateApplicationStatus(APPSTATUS_INIT_TRACKER);
              break;
```

Frammento 4.4.1: Metodo di inizializzazione (parte 1 di 3)

```
      case APPSTATUS_INIT_TRACKER:
            // carica il database e le informazioni sui tracker
            try
            {
                  mLoadTrackerTask = new LoadTrackerTask();
                  // se execute va a buon fine procedi con è
                  // APPSTATUS_INITED
                  mLoadTrackerTask.execute();
            }
            catch (Exception e)
            {
                  DebugLog.LOGE("Loading tracking data set failed");
            }
            break;

      case APPSTATUS_INITED:

            System.gc();

            // funzione nativa di post iniz<ializzazione
            nQCARInitializedNative();

            // Tempo di caricamento dell'applicazione
            long splashScreenTime = System.currentTimeMillis() -
                                          mSplashScreenStartTime;
            long newSplashScreenTime = 0;
            if (splashScreenTime < MIN_SPLASH_SCREEN_TIME)
            {
                  newSplashScreenTime = MIN_SPLASH_SCREEN_TIME -
                  splashScreenTime;
            }

            Handler handler = new Handler();
            handler.postDelayed(new Runnable() {
                  public void run()
            {
                  // nascondi la pagina di inizializzazione
                  mSplashScreenView.setVisibility(View.INVISIBLE);

                  // attiva il render
                  mRenderer.mIsActive = true;

                  // aggiungi la GLSurfaceView prima di iniziare la
                  // cattura dei frame
                  addContentView(mGlView, new LayoutParams(
                                          LayoutParams.FILL_PARENT,
                                          LayoutParams.FILL_PARENT));

                  // inizia la cattura dei frame
                  updateApplicationStatus(APPSTATUS_CAMERA_RUNNING);
            }
            }, newSplashScreenTime);

            break;
```

Frammento 4.4.2: Metodo di inizializzazione (parte 2 di 3)

```
        case APPSTATUS_CAMERA_STOPPED:
            // richiama la funzione nativa di fine riprese
            stopCamera();
            break;

        case APPSTATUS_CAMERA_RUNNING:
            // richiama la funzione nativa di inizio riprese
            startCamera();
            break;

        default:
            throw new RuntimeException("Invalid application state");
    }
}
```

Frammento 4.4.3: Metodo di inizializzazione (parte 3 di 3)

ImageTargetRenderer.java contiene un sola classe omonima che possiede tutti gli strumenti per il rendering dell'oggetto. Implementa l'interfaccia `GLSurfaceView.Rendered` e quindi i seguenti tre metodi.

- `public void onSurfaceCreated(GL10 gl, EGLConfig config)` è il metodo che viene eseguito dopo la creazione di un'istanza della classe `GLSurfaceView.Rendered` o di una classe figlia. Nel caso specifico dell'esempio questo metodo viene utilizzato per richiamare il metodo nativo `initRendering()` dove viene creata la struttura OpenGL dell'oggetto virtuale; successivamente si esegue una chiamata al metodo della classe QCAR `onSurfaceCreated()`;
- `public void onSurfaceChanged(GL10 gl, int width, int height)` questo metodo viene chiamato quando la dimensione della superficie di visualizzazione cambia, richiama la funzione nativa `updateRendering(width, height)` e successivamente il metodo della classe QCAR `onSurfaceChanged(width, height)`;
- `public void onDrawFrame(GL10 gl)` viene chiamato per disegnare su un frame, verifica innanzitutto che il renderer sia attivo e successivamente richiama la funzione nativa `renderFrame()`.

Il file *QCARSampleGLView.java* contiene la classe con lo stesso nome che estende `GLSurfaceView` e ha il compito di inizializzare la superficie nella quale si visualizzeranno i frame, all'interno di questa classe sono contenute altre due classi private che servono per creare un contesto personalizzato e una classe per scegliere quale configurazione utilizzare.

Il file *Texture.java* racchiude la classe omonima che serve per caricare la struttura dell'oggetto da visualizzare. Al suo interno è implementato il metodo statico `loadTextureFromApk(String fileName, AssetManager assets)` che restituisce un oggetto di tipo *Texture* e il metodo `getData()` che restituisce i dati della struttura.

4.4.2 Codice nativo dell'esempio

I file che compongono il codice nativo dell'esempio sono essenzialmente sette e altri due file sono quelli di istruzioni per la compilazione. I file che costituiscono il codice nativo sono i seguenti.

- *CubeShaders.h*: è un file header nel quale si definiscono gli OpenGL shaders[5] per poter rendere più efficiente l'utilizzo delle librerie;
- *ImageTarget.h, Texture.h* e *SampleUtils.h*: sono gli header file che contengono le definizioni dei metodi che sono implementati rispettivamente nei file *ImageTarget.cpp, Texture.cpp* e *SampleUtils.cpp*;
- *ImageTarget.cpp*: è il file principale di codice nativo nel quale sono implementate tutte le funzioni che nel codice Java sono dichiarate con la parola chiave *native*;
- *SampleUtils.cpp*: contiene le implementazioni delle funzioni di supporto, come il calcolo del prodotto tra matrici, la traslazione della posizione, e la rotazione dell'oggetto virtuale;
- *Teapot.h*: contiene il disegno della teiera che verrà costruita nel processo di rendering;
- *Texture.cpp*: le funzioni implementate in questo file hanno lo scopo di rendere accessibili nel codice C++ i dati della struttura dell'oggetto virtuale, caricata da file nella classe *Texture.java.*

Nel file *ImageTarget.cpp* è contenuta la funzione nativa principale dell'applicazione cioè quella che si occupa del rendering dell'immagine. Questa funzione è suddivisa in tre blocchi principali:

- preparazione delle librerie grafiche e inizio del rendering (v. frammento di codice 4.4.4);
- tracciamento del trackable e caricamento della struttura da usare, nel nostro caso quella della teiera (v. frammento di codice 4.4.5);

5 Insieme di istruzioni per la costruzione dell'oggetto virtuale

- rendering dell'oggetto (v. frammento di codice 4.4.6).

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// Render video background:
QCAR::State state = QCAR::Renderer::getInstance().begin();

glEnable(GL_DEPTH_TEST);
glEnable(GL_CULL_FACE);
```

Frammento 4.4.4: Preparazione all'utilizzo delle librerie OpenGL ES e inizio del rendering

All'interno di un ciclo *for*, vengono eseguiti i frammenti di codice 4.4.5 e 4.4.6 per ogni trackable trovato.

```
const QCAR::Trackable* trackable = state.getActiveTrackable(tIdx);
QCAR::Matrix44F modelViewMatrix =
             QCAR::Tool::convertPose2GLMatrix(trackable->getPose());

// Scelta della struttura correlata all'immagine tracciata
int textureIndex = (!strcmp(trackable->getName(), "stones")) ? 0 : 1;
const Texture* const thisTexture = textures[textureIndex];
//animazione della teiera
animateTeapot(modelViewMatrix);
```

Frammento 4.4.5: creazione del trackable e caricamento della struttura collegata

```
QCAR::Matrix44F modelViewProjection;

SampleUtils::translatePoseMatrix(0.0f, 0.0f, kObjectScale,
                                                 &modelViewMatrix.data[0]);
SampleUtils::scalePoseMatrix(kObjectScale, kObjectScale, kObjectScale,
                                       &modelViewMatrix.data[0]);
SampleUtils::multiplyMatrix(&projectionMatrix.data[0],
                            &modelViewMatrix.data[0] ,
                            &modelViewProjection.data[0]);

glUseProgram(shaderProgramID);

glVertexAttribPointer(vertexHandle, 3, GL_FLOAT, GL_FALSE, 0,
                                (const GLvoid*) &teapotVertices[0]);
glVertexAttribPointer(normalHandle, 3, GL_FLOAT, GL_FALSE, 0,
                                (const GLvoid*) &teapotNormals[0]);
glVertexAttribPointer(textureCoordHandle, 2, GL_FLOAT, GL_FALSE, 0,
                                (const GLvoid*) &teapotTexCoords[0]);

glEnableVertexAttribArray(vertexHandle);
glEnableVertexAttribArray(normalHandle);
glEnableVertexAttribArray(textureCoordHandle);

glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, thisTexture->mTextureID);
glUniformMatrix4fv(mvpMatrixHandle, 1, GL_FALSE,
                          (GLfloat*)&modelViewProjection.data[0] );
glDrawElements(GL_TRIANGLES, NUM_TEAPOT_OBJECT_INDEX, GL_UNSIGNED_SHORT,
                       (const GLvoid*) &teapotIndices[0]);

SampleUtils::checkGlError("ImageTargets renderFrame");
```

Frammento 4.4.6: Rendering dell'oggetto

Quando la funzione ha termine si devono disabilitare tutti gli elementi attivati (v. frammento di codice 4.4.7).

```
glDisableVertexAttribArray(vertexHandle);
glDisableVertexAttribArray(normalHandle);
glDisableVertexAttribArray(textureCoordHandle);

QCAR::Renderer::getInstance().end();
```

Frammento 4.4.7: rilascio risorse e blocco del rendering

Come si può vedere l'esempio è abbastanza completo. Una modifica interessante è stata l'animazione dell'oggetto virtuale: questa è stata realizzata agendo sulla matrice *modelViewMatrix* utilizzando una funzione di supporto (v. frammento 4.4.8).

```
void
animateTeapot(QCAR::Matrix44F& modelViewMatrix)
{
      static float rotateBowlAngle = 0.0f;
      static float moveXY = 0.0f;
      static double prevTime = getCurrentTime();
      double time = getCurrentTime();

      float dt = ((float)(time-prevTime))/10.0f;

      rotateBowlAngle += dt * 18000.0f/3.1415f;

      moveXY += dt;
      SampleUtils::translatePoseMatrix(50.0f*cos(2.0f*3.1415f*moveXY),
         50.0f*sin(2.0f*3.1415f*moveXY), 0.0f, &modelViewMatrix.data[0]);
      SampleUtils::rotatePoseMatrix(rotateBowlAngle, 0.0f, 0.0f, 1.0f,
                                    &modelViewMatrix.data[0]);

      prevTime = time;
}
```

Frammento 4.4.8: Funzione per l'animazione della teiera

La funzione deve essere invocata subito dopo aver convertito la matrice di posizione (v. frammento di codice 4.4.5). Questa funzione agisce sulla teiera facendole eseguire la rotazione su se stessa e la rivoluzione attorno al centro dell'immagine.

```
<?xml version="1.0" encoding="UTF-8"?>
<QCARConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="qcar_config.xsd">
  <Tracking>
   <ImageTarget size="247 173" name="stones" />
   <ImageTarget size="247 173" name="chips" />
  </Tracking>
</QCARConfig>
```

Frammento 4.4.9: file `config.xml` originale

```
<?xml version="1.0" encoding="UTF-8"?>
<QCARConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="qcar_config.xsd">
  <Tracking>
   <ImageTarget name="stones" size="247 173">
      <VirtualButton name="muovi" rectangle="-108.68 -53.52 -75.75
                  -65.87" enabled="true"></VirtualButton></ImageTarget>
  </Tracking>
</QCARConfig>
```

Frammento 4.4.10: file `config.xml` modificato con l'aggiunta del pulsante virtuale e la rimozione dell'immagine "chips" dalla lista dei trackable

Una modifica più profonda è stato l'inserimento di un pulsante virtuale "invisibile", ovvero non disegnato nell'immagine, che eseguisse la stessa animazione precedente solamente dopo la sua occlusione; per fare ciò si è intervenuti, oltre che nel codice, anche nel file `config.xml` originale (v. frammento di codice 4.4.8) aggiungendo il pulsante virtuale associato all'immagine "*stone*" e rimuovendo dalla lista dai possibili

trackable l'immagine "*chips*"(v. frammento di codice 4.4.9).Parallelamente l'intervento sul codice è stato significativo (v. frammento di codice 4.4.10): non esisteva più l'esigenza di eseguire il ciclo *for* per rilevare tutti i possibili trackable ma bastava verificare se fosse stata tracciata l'immagine predefinita, e successivamente valutare l'occlusione del pulsante virtuale.

```
//valuta se ci sono trackable attivi
if (state.getNumActiveTrackables()>0)
    {
        //crea l'oggetto trackable e restituiscine la posizione
        const QCAR::Trackable* trackable = state.getActiveTrackable(0);
        QCAR::Matrix44F modelViewMatrix =
            QCAR::Tool::convertPose2GLMatrix(trackable->getPose());

       //verifica che il trackable sia di tipo image target in caso
       // affermativo fai il cast dell'oggetto Trackable a ImageTarget
        assert(trackable->getType() == QCAR::Trackable::IMAGE_TARGET);
        const QCAR::ImageTarget* target = static_cast<const
                                        QCAR::ImageTarget*> (trackable);

        // crea l'oggetto Virtual Button se esiste altrimenti
       //inizializzalo a null
        const QCAR::VirtualButton* button = target->getVirtualButton(0);

         //se il bottone non è null vedi se è premuto
        if (button != NULL)
        {
            // se è premuto anima la teiera
            if (button->isPressed())
            {
             animateTeapot(modelViewMatrix);
            }
        }
//esegui impostazione e disegna l'oggetto
```

Frammento 4.4.11: Nuovo frammento di codice nella funzione renderFrame() del codice nativo

Il rimanente codice resta invariato.

4.5 Conclusioni

In questo capitolo è stato visto come sia possibile creare un'applicazione di AR utilizzando QCAR SDK, osservando come un buon punto di partenza sia la modifica degli esempi forniti direttamente da Qualcomm, che oltre all'applicazione analizzata e modificata fornisce altri esempi anche più complessi per poter capire meglio il funzionamento di tutti gli elementi dell'SDK.

Bibliografia

[1] it.wikipedia.org/wiki/Realtà_aumentata

[2] http://en.wikipedia.org/wiki/Augmented_reality#Software

[3] Vallino James. *Interactive augmented reality*. PhD thesis, Department of Computer Science, University of Rochester, USA (1998).

[4] P. Milgram, H. Takemura et al. *Augmented Reality: A Class of Displays on the Reality-Virutality Continuum.* In Proceedings of SPIE: Telemanipulatorand Telepresence Technologies 2351 (1994), 282–292.

[5] Stilman M., Michel P., Chestnutt J., Nishiwaki K., Kagami S., Kuffner J.J. (2005). *Augmented Reality for Robot Development and Experimentation*. Tech. Report CMU-RI-TR-05-55, Robotics Institute, Carnegie Mellon University.

[6] Bimber, O., Raskar, R.. *Modern Approaches to Augmented Reality.* 25th Annual Conference of the European Association for Computer Graphics, Interacting with Virtual Worlds, Tutorial 8, pp. 1-86, 2004

[7] Bell, B., Feiner, S., Höllerer, T.. *Information at a Glance.* IEEE Computer Graphics and Applications, vol. 22, n. 4, Jul/Aug, pp. 6-9, 2002.

[8] Pinz, A., Brandner, M., Ganster, H., Kusej, A., Lang, P., Ribo, M., *Hybrid Tracking in Augmented Reality*, ÖGAI Journal, vol. 21, n. 1, pp. 17-24, 2000.

[9] M. Fischler and R. Bolles, *Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography.* Communications ACM, vol. 24, no. 6, pp. 381–395, 1981.

[10] Wayne Piekarski. *Interactive 3d modelling in outdoor augmented reality worlds*. Bachelor of Engineering in Computer Systems Engineering (Hons), University of South Australia, Adelaide 2004.

[11] Ruobing Yang, *The Study of improvment of Augmented Reality based on future matching,* Media School, Linyi University, IEEE, pp586-589, 2011

[12] Li Yi-bo, Kang Shao-peng, Qiao Zhi-hua, Zhu Qiong, *Development Actuality and Application of Registration Technology in Augmented Reality,* Department of Auto-control, Shenyang Institute of Aeronautical Engineering, Shenyang, International Symposium on Computational Intelligence and Design, IEEE, pp.69-74, 2008

[13] Piekarski W, Thomas H. B., *Interactive Augmented Reality Techniques for Construction at a Distance of 3D Geometry,* The Eurographics Association, 2003

[14] James R Vallino, *Interactive Augmented Reality*, Ph Thesis, 2008

[15] Yen-Hsu Chen, Tsorng-Lin Chia, *A Vision-Based Augmented-Reality System For Multiuser Collaborative Environments,* IEEE TRANSACTIONS ON MULTIMEDIA, VOL. 10, NO. 4, pp 585-595,2008

[16] https://developer.qualcomm.com/develop/mobile-technologies/augmented-reality

[17] http://developer.android.com/index.html

Indice delle illustrazioni