Alma Mater Studiorum
Università di Bologna

Facoltà di Ingegneria II

Corso di Laurea Specialistica in Ingegneria Informatica

Tesi di Laurea in Fondamenti di Computer Graphics

Progetto e sviluppo di un game che utilizza la realtà aumentata su piattaforma Android

Candidato:
Andrea Novaga

Relatore:
Prof.ssa Serena Morigi

Correlatore:
Ludovico Cellentani

Anno Accademico 2010/2011 - Sessione III

Indice

Introduzione

Fra le diverse tecniche, inerenti al campo della *computer graphics*, della *computer vision* o dei *sistemi multimediali* in generale, negli ultimi anni ha preso piede la **realtà aumentata**. Tramite la realtà aumentata, abbreviata anche **AR** (*augmented reality*), è sostanzialmente possibile comporre, immagini - reali, cioè prese direttamente dalla realtà tramite un qualunque dispositivo di tipo telecamera, con immagini *virtuali*, ossia realizzate tramite tecniche di computer grafica, arrivando a creare dei risultati davvero sorprendenti.

I campi in cui tutt'ora questa tecnica ha trovato applicazione sono molteplici, e vanno da quello educativo, a quello medico, alla navigazione e orientamento, fino ovviamente a quello dell'intrattenimento. Fra le diverse idee, basti ad esempio pensare alla possibilità di inquadrare un qualunque tipo di opera in un museo ottenendone al contempo (sullo stesso schermo sul quale viene ripresa l'immagine) tutte le informazioni a riguardo, oppure a quella di utilizzare un particolare navigatore che mostra in sovra-impressione i nomi degli edifici e delle vie che vengono inquadrati. Il settore del *gaming* in particolare, è uno di quelli in cui l'AR sta recentemente trovando molte nuove ed originali implementazioni, tant'è che è possibile trovare diversi giochi di questo tipo anche fra le grandi produzioni.

Nonostante i molteplici contesti applicativi tuttavia la diffusione di questa tecnica è stata per un determinato periodo limitata dai requisiti tecnici richiesti ad un dispositivo per poterla supportare, quali la cattura dell'immagine tramite videocamera e la capacità di elaborare immagini virtuali, in primis, oppure altri assai più specifici, come ad esempio la presenza di GPS o di bussola, per realizzare funzionalità particolari (si pensi all'idea del navigatore interattivo). Inizialmente veniva infatti pensata per essere utilizzata esclusivamente su home-computers o su dispositivi particolari, ad hoc per tale

funzionalità.

Recentemente si è verificata una rapida diffusione dei moderni dispositivi portatili, quali ad esempio smartphones o tablets di nuova generazione, le caratteristiche dei quali si differenziano notevolmente da quelle dei tradizionali telefoni cellulari, non solo per la mera potenza di calcolo e per tutte le funzionalità aggiunte, ma anche per l'elevato grado di configurabilità e la facilità di sviluppo e distribuzione di nuove applicazioni. In questo contesto l'AR ha trovato un terreno assai più congeniale per la sua diffusione. Completa gestione della videocamera e delle immagini ottenute, elaborazione di computer graphics, e tutte le altre funzionalità del caso quali GPS, bussola, accelerometro, etc., sono tutte caratteristiche ormai offerte da qualunque di questi moderni dispositivi. Tornando nuovamente all'esempio museo, si pensi alla comodità per un utente di utilizzare la suddetta funzionalità di AR per vedere tutte le informazioni delle opere d'arte direttamente dal proprio smartphone e tramite una semplice *app* (come vengono generalmente chiamate le applicazioni su dispositivi mobile).

Con la presente tesi si intende realizzare un'applicazione di gaming che si basa su AR, su una piattaforma software particolarmente diffusa, disponibile per smartphones e tablets, chiamata Android, della quale si parlerà in seguito e se ne analizzeranno le caratteristiche, tanto approfonditamente quanto necessario. Per la precisione si tratterà di un semplice game, dalle strategie non troppo complicate ma che sfrutta la realtà aumentata anche come elemento del game-play stesso. Non ci si concentrerà comunque in maniera eccessiva sugli aspetti relativi allo specifico game-play, poiché certamente non sono d'interesse per la presente tesi, quanto piuttosto su tutti quegli aspetti che sono senza dubbio fondamentali per questo tipo di applicazione, quali per esempio prestazioni o tempestività nella risposta a determinati input.

Ad ogni modo, a prescindere dallo specifico programma, è di nostro interesse definire prima un sistema generico per la gestione delle funzionalità dell'AR stessa, sopra al quale, seguendo uno schema di progettazione su più livelli, il gioco vero e proprio verrà poi opportunamente realizzato. Potremmo in vero definire questo sistema base come una sorta di *ambiente di sviluppo* o *framework* per la realizzazione di applicazioni (in particolare di gaming) su

Android e facenti uso della realtà aumentata. Questa scelta in vero ci aiuterà anche a comprendere meglio quali sono le problematiche in comune coinvolte nel processo di sviluppo di una qualunque applicazione di questo tipo.
E' stato deciso di denominare l'intera applicazione di progetto **ARGame**, nome con il quale di qui in avanti verrà chiamata.
Nel corso della tesi verranno descritte con precisione tutte le diverse fasi nelle quali è stato articolato l'intero processo di sviluppo dell'applicazione, quali analisi, progettazione ed infine valutazione della qualità del risultato finale e delle prestazioni, proponendo se possibile anche eventuali miglioramenti. Si dedicherà inoltre un approfondimento alle varie tematiche affrontate, a partire dal concetto e dalle proprietà stesse della realtà aumentata, alla caratteristiche del gaming in generale, fino agli aspetti principali e d'interesse di Android. Allo stesso modo si porrà una particolare attenzione alle maggiori problematiche che riguardano lo sviluppo di un'applicazione di questo tipo su tale piattaforma e le risoluzioni possibili. Lo scopo infatti, è anche quello di saggiare, almeno in parte, le potenzialità del sistema operativo in questione, il quale si sta sempre più confermando come uno dei due leader indiscussi nel settore mobile.

Capitolo 1

Realtà Aumentata

Per definizione, la realtà aumentata è *la sovrapposizione di livelli informativi (elementi virtuali e multimediali, dati geo-localizzati, ecc.) all'esperienza reale di tutti i giorni*. L'aggettivo aumentata sta ad indicare che di fatto quel che si effettua è un aumento della realtà per come essa viene percepita, attraverso l'aggiunta di altri contenuti multimediali di qualunque tipo, quali ad esempio scritte, immagini o qualunque tipo di oggetto virtuale modellato in grafica 3D. Da sottolineare che per realtà percepita si intende propriamente la realtà *in movimento* e non quella ad esempio di immagini statiche, ragion per cui le tecniche che riguardano l'intero processo di AR vanno effettuate in tempo reale.

In riferimento alla Fig. reffig:AR schema 1 e alla Fig. reffig:AR schema 2, il processo AR si basa sostanzialmente sulla cattura delle immagini da parte di un qualunque dispositivo di ripresa, quale ad esempio la fotocamera/videocamera di un cellulare, sulla localizzazione ed elaborazione degli elementi virtuali da visualizzare, in base al contenuto dell'immagine stessa o di altri input (blocco CV in Fig. 1.2), sulla loro composizione con l'immagine reale e sull'invio dell'immagine così elaborata sul display (blocco CG). Il procedimento che può risultare semplice richiede in realtà l'applicazione di sofisticate tecniche in campo di visione artificiale, elaborazione dell'immagine e computer grafica, generalmente strettamente dipendenti dalle specifiche funzionalità dell'applicazione che s'intende realizzare, e spesso richiede anche l'ausilio di sensori fisici particolari. Il blocco di elaborazione CV ottiene in input l'immagine reale, la quale viene posta come sfondo della scena, pro-

ducendo lo stesso identico effetto che si avrebbe con qualunque semplice effetto di ripresa video, mentre sopra di essa vengono visualizzati una serie di contenuti *virtuali* ossia ottenuti secondo qualunque tecnica di elaborazione grafica, in 2D o in 3D. Il modo in cui si determina quali di queste informazioni debbano essere presenti ed in che modo debbano essere visualizzate, dipende dalla particolare applicazione in questione: può essere che dipenda ad esempio dal contenuto dell'immagine stessa, ed in tal caso si applicano generalmente tecniche di elaborazione dell'immagine e di *template-matching*, oppure da sorgenti d'informazione esterne, ed in questo caso si applicano differenti procedure.
In ogni caso è chiaro che la natura dell'informazione dell'immagine presa direttamente da videocamera è generalmente differente dal quella delle immagini elaborate virtualmente, per questo particolari tecniche devono essere adottate per adattare le due tipologie ad un contesto comune e visualizzare così l'immagine finale, ottenuta dalla loro combinazione, su schermo. Ovviamente queste tecniche dipendono generalmente dallo specifico contesto applicativo in questione. Un esempio piuttosto comune, abbozzato in Fig. 1.3, è il caso in cui gli elementi virtuali da visualizzare siano oggetti 3D renderizzati tramite librerie OpenGL o DirectX: la tecnica comunemente utilizzata è quella di porre il contenuto dell'immagine reale in una texture e applicare a sua volta questa ad una superficie di sfondo, la quale viene renderizzata prima di qualunque altro elemento.
Lo schema in Fig. 1.1 riassume brevemente i passi che compongono il ciclo di processo di AR.
In Fig. 1.2 viene illustrato il processo di AR dal punto di vista del flusso informativo.
Da questa si può notare come l'intero processo elaborativo sia sostanzialmente diviso in due blocchi distinti. Le possibili operazioni eseguite all'interno del blocco CV (per le operazioni di *computer vision*) sono:

- analizzare il contenuto dell'immagine ottenuta da telecamera
- effettuare su questa le dovute procedure di conversione o image-processing
- attuare le procedure di pattern-recognition per l'individuazione di particolari forme o materiali relative a marker o particolari oggetti

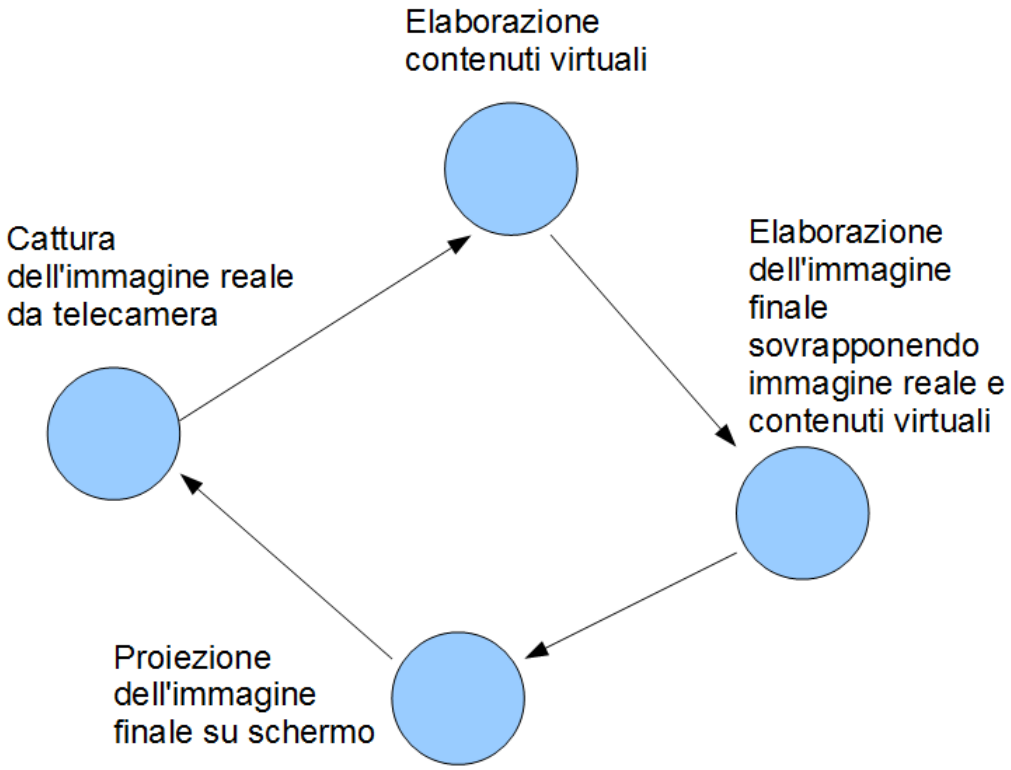

Figura 1.1: schema del ciclo di funzionamento del processo di AR

- integrare le informazioni così ottenute con altre provenienti da particolari fonti o sensori, come ad esempio informazioni di geo-localizzazione ottenute da GPS
- ottenere il posizionamento della telecamera di vista nel mondo rispetto ad un determinato sistema di coordinate

Le informazioni così ottenute sono poi passate al blocco CG (operazioni di *Computer Graphics*), il quale invece si occupa di:

- mostrare il contenuto dell'immagine da telecamera come sfondo dell'intera scena virtuale (generalmente ciò viene effettuato per mezzo di un'apposita texture)

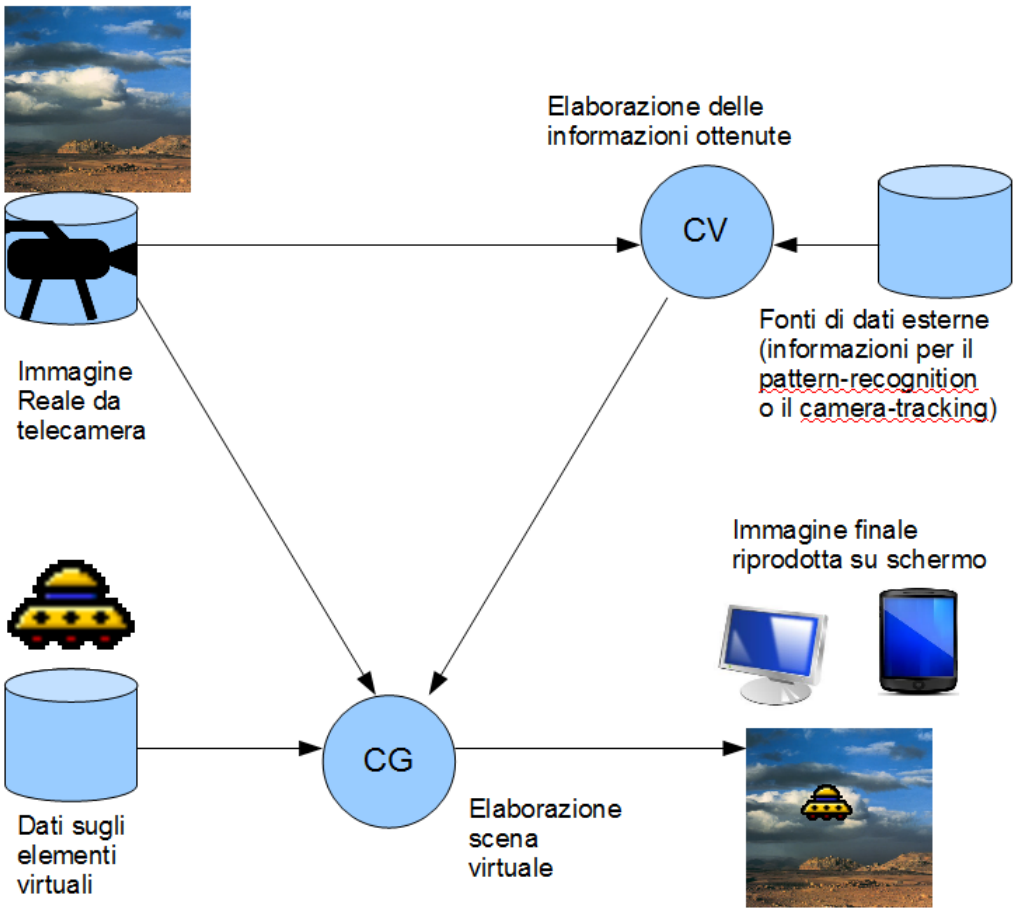

Figura 1.2: schema del flusso informativo del processo AR. Da notare in particolare la presenza dei due diversi blocchi di elaborazione, CV (*Computer Vision*) e CG (*computer graphics*).

- elaborare quali elementi virtuali posizionare in scena, sulla base delle informazioni ottenute
- implementare eventuali effetti di rendering, sempre sulla base del contenuto dell'immagine reale
- comporre il tutto nel rendering finale su schermo

In una normale registrazione video il flusso informativo andrebbe direttamente dal dispositivo di cattura a quello di visualizzazione, ciò che avviene in questo caso è invece un'interruzione di tale flusso, il quale passa ora attraverso tutta quella fase che riguarda il recupero ed il rendering dei contenuti

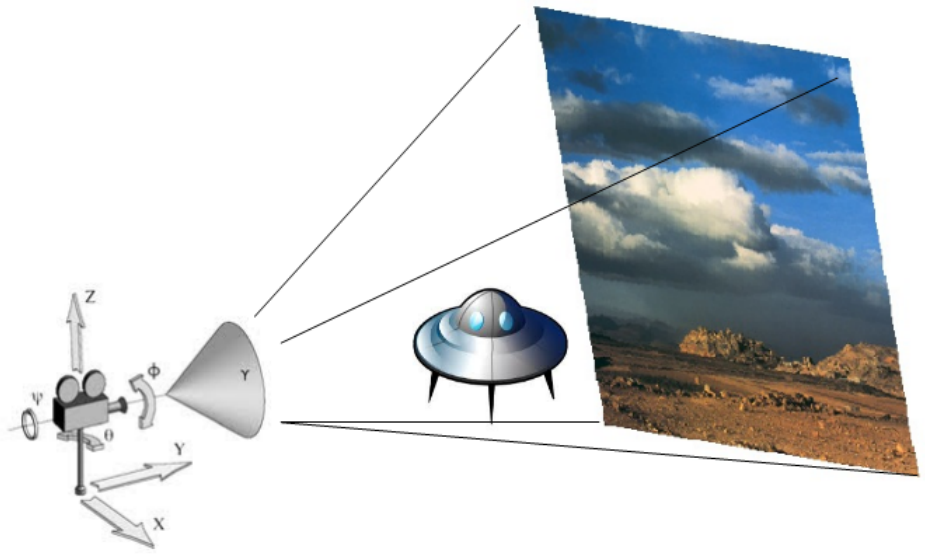

Figura 1.3: Possibile metodo di elaborazione dell'immagine finale nel contesto dell'AR: elementi virtuali renderizzati sopra una superficie con una texture riportante l'immagine reale. Da notare come tale superficie venga proiettata perpendicolarmente davanti alla telecamera virtuale.

aggiuntivi. Tutto ciò va inevitabilmente ad incidere con i tempi di visualizzazione (in pratica il frame-rate) dell'immagine finale, aspetto che però rimane generalmente trascurabile, almeno finché non viene avvertito dalla parte della percezione visiva dell'utente. Parlando chiaramente, il tipo di realtà che l'AR va ad aumentare è la realtà in movimento percepita, quindi le immagini che si susseguono nel loro complesso per generare il video che verrà visualizzato su schermo, ragion per cui anche la gestione del tempo è importante. I processi di elaborazione e inserimento delle informazioni vanno dunque effettuati in tempo reale, con tutte le conseguenze che cioè comporta sia sulla potenza minima richiesta al dispositivo che sull'ottimizzazione delle relative tecniche.

Particolarmente critica può essere la parte relativa all'identificazione e localizzazione delle informazioni virtuali da visualizzare, che può variare a seconda del particolare contesto e può avvenire in differenti modi. Generalmente si possono distinguere due casi: informazioni ricavate direttamente dal contenuto dell'immagine catturata, o informazioni ottenute invece in maniera

indipendente da queste. Ovviamente per questo secondo caso ci si aspetta una particolare cura nel modo in cui tali informazioni vengono ricavate, poiché deve comunque esserci coerenza fra queste informazioni aggiunte e l'immagine reale. Si pensi ad esempio ad un navigatore che mostra nomi di vie o edifici che vengono ripresi con la telecamera e che per far cioè si avvalga delle funzionalità di geo-localizzazione, per ricavare la propria posizione sulla mappa, e la bussola per capire verso dove ci si orienti e che cosa si ha di fronte. E' in questo caso indispensabile il perfetto funzionamento ed utilizzo di questi strumenti per ottenere il risultato desiderato, dal momento che non c'è corrispondenza diretta fra le informazioni da questi fornite e quelle direttamente ricavabili dall'immagine reale. Il vantaggio di questo tipo di approccio con la realtà aumentata è che risulta computazionalmente meno oneroso rispetto all'altro caso, a fronte di una buona efficienza delle strumentazioni di supporto.

Figura 1.4: esempio di utilizzo di realtà aumentata su device iPhone: si tratta di un comune GPS che utilizza funzionalità di geo-localizzazione e bussola ed evidenzia i nomi di vie ed edifici inquadrati

Per informazioni che devono essere invece ricavate direttamente dall'immagine che è stata catturata, ossia direttamente dal contesto davanti all'osservatore, ci si avvale in genere di tecniche di *pattern recognition* (riconoscimento di forma), per riconoscere specifiche forme bidimensionali, tipi di oggetti o di superfici, a seconda di cioè che si intende identificare ai fini dell'applicazione.

Figura 1.5: altro esempio di un comune sistema di navigazione con AR, questa volta per dispositivo Android

Queste sono a loro volta basate su differenti tipi di algoritmi di riconoscimento, spesso anche piuttosto complessi e computazionalmente onerosi. In questa categoria si possono ulteriormente identificare altri due casi particolari: tecniche di riconoscimento basate su marker e quelle non basate su marker. Chiariamo a questo punto il concetto di **marker**: come dice il nome stesso (che in inglese significa appunto marchio o identificatore), si tratta di una particolare figura, generalmente stampata su superficie piana (ma non necessariamente), e fatta in modo tale da poter essere riconosciuta tramite un determinato algoritmo, una volta inquadrata nell'immagine. Esempi di markers sono illustrati in Fig. 1.6.

Un esempio di applicazione marker-based AR viene invece dato in Fig. 1.7.

Quello che può interessare può essere semplicemente sapere se tale marker è presente o meno all'interno dell'immagine, oppure la sua posizione all'interno dell'immagine stessa, oppure ancora, se il marker possiede delle proprietà particolari, quali ad esempio il suo collocamento in uno spazio tridimensionale, la sua rotazione, etc. Salvo determinate informazioni aggiuntive che si intendono ottenere, gli algoritmi di pattern recognition per questo particolare caso sono comunque relativamente semplici, proprio perché il tipo di informazione che si vuole riconoscere è in realtà già ben determinato a priori, e fra le due (marker e marker-less) questa è generalmente la tecnica più semplice

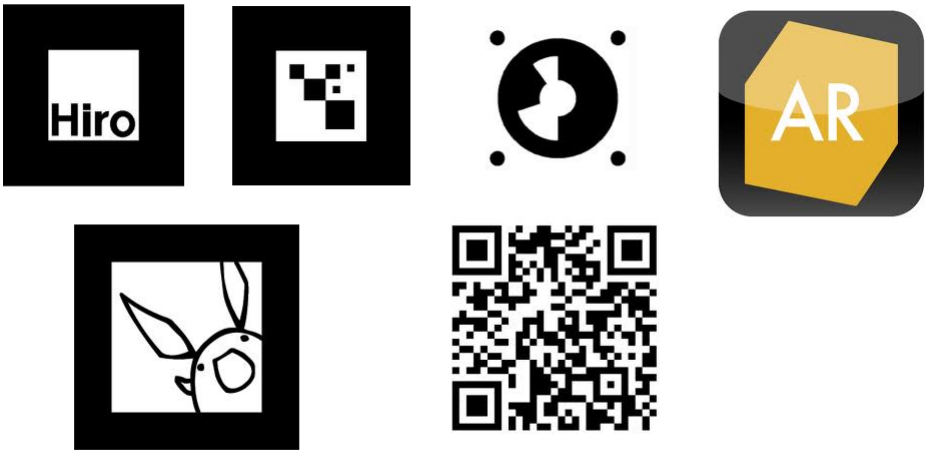

Figura 1.6: esempi di markers utilizzabili per applicazioni di realtà aumenta. In genere un marker è esclusivamente in bianco e nero, ma possono essere anche utilizzati markers a colori.

e meno onerosa da implementare.
Tecniche di pattern recognition che non si basano sull'utilizzo di markers sono invece generalmente più complicate, o comunque richiedono assai più attenzione nella definizione dell'informazione da riconoscere, a meno che non si tratti di un tipo d'informazione di tipo elementare. Non sono molti infatti i casi per cui, dato un certo tipo di oggetto, è possibile identificare le caratteristiche comuni a tutti gli oggetti di quel tipo. Uno dei pochi esempi (piuttosto comune a dire il vero) può essere il riconoscimento di certi tratti del volto umano, quali in particolar modo la bocca o gli occhi, o di altre parti del corpo umano o l'intera fisionomia di questo.

1 Applicazioni della realtà aumentata

Per via dei requisiti necessari per il suo utilizzo, sia come funzionalità che come potenza di calcolo, la tecnica dell'AR ha avuto una diffusione assai limitata all'inizio, ed è stata utilizzata al più in contesti particolari, quali ad esempio in campo di ricerca medica o militare, talvolta in quello educativo, o comunque limitata a sistemi di tipo home-pc, sui quali però si può dire non avere mai trovato vere applicazioni degne d'interesse e di facile diffusione.

Figura 1.7: applicazione di base compresa fra i software pre-installati all'interno di una console Nintendo 3DS: si tratta di un semplice esempio di implementazione di realtà aumentata basata su markers, che in questo caso sono delle semplici card collezionabili

Con le nuove tecnologie tuttavia, in particolare con i dispositivi mobile di ultima generazione quali smartphone, tablet o console portatili, nuovi scenari son stati possibili nei quali l'AR sembra aver trovato il terreno ideale. A contribuire a cioè è soprattutto il supporto che questi nuovi dispositivi riescono a dare per tutte quella funzionalità che si rendono necessarie per l'implementazione della realtà aumentata. La natura stessa dei device portatili sembra inoltre dare un rilevante stimolo all'immaginazione degli sviluppatori: un moderno smartphone od un tablet infatti, potrebbe sostituirsi benissimo all'uso ad esempio di un navigatore o di qualunque altro dispositivo portatile ad uso particolare. Basti pensare ancora all'esempio del lettore delle informazioni delle opere in un museo, utilizzabile da qualunque utente sul proprio smarphone, anzichè su un qualche tipo di dispositivo apposito consegnato separatamente.

Oltre ai vari contesti di utilità, come può ad esempio essere l'idea del navigatore, la realtà aumentata sta riscuotendo anche un certo successo nell'ambito

del puro *entertainment*, in particolare nel *gaming*. Il vantaggio dell'AR in questo contesto è sicuramente il fattore novità portato che spinge anche a pensare a nuove possibili tipologie di game-play e di conseguenza anche l'attrattiva che è in grado di esercitare sul pubblico. Diversi nuovi giochi sono stati recentemente ideati sulle più svariate piattaforme, sia per smartphones e tablet quali appunto iOS ed Android, ma anche per console portatili e home-console, interamente incentrati sull'idea di realtà aumentata. Alcuni di questi, nonostante siano ancora da considerare più come una sorta di sperimentazione verso questa nuova frontiera, hanno budget paragonabili a quelli di grandi produzioni. Fra questi possiamo citare gli esempi in Fig. 1.7, Fig. 1.10 o Fig. 1.11.

Figura 1.8: un'altra applicazione fra quelle disponibili su un dispositivo Nintendo 3DS, in questo caso si tratta di un esempio di realtà aumentata non basata sull'utilizzo di markers

In [11], [12] e [13] abbiamo inoltre qualche approfondimento maggiore su particolari progetti che riguardano questi tipi di games, illustrati anche in Fig. 1.12 e Fig. 1.13. Da quest'ultimo in particolare è stata trovata l'idea del gioco finale per come sarè poi sviluppato.

Figura 1.9: un semplice esempio di realtà aumentata basata sul riconoscimento facciale

Figura 1.10: un video-game per il sistema XBox360 Kinect: un esempio di realtà aumentata basata sul riconoscimento del corpo

Figura 1.11: Invisibles, un esempio di sofisticato game disponibile su console psp, che utilizzaza la realtà aumentata

Figura 1.12: Il classico gioco da tavolo del Monopoli reso più interessante tramite l'utilizzo della realtà aumentata

Figura 1.13: AR Tower Defense, un videogioco sulla realtà aumentata basata sull'utilizzo di markers. Si è tratta particolare aspirazione da questo game per l'ideazione dell'applicazione di progetto.

Capitolo 2

Elementi del gaming

L'industria del gaming esiste ormai da diverso tempo ed tutt'oggi uno dei settori più in fermento in assoluto in ambito di tecnologia ed intrattenimento. Ormai moltissimi tipi di videogiochi sono stati ideati, dalle caratteristiche più differenti e per i sistemi più disparati, e nuove idee continuano ancora ad emergere. L'utilizzo della realtà aumentata nel contesto del gaming rientra sicuramente nella cerchia di novità d'interesse, per quanto sembra che, per il momento, sia riuscito ad affermarsi solo in maniera marginale, probabilmente per la sua eccessiva particolarità. Effettivamente riuscire a trovare il modo di sfruttare l'idea base dell'AR in maniera significativa ai fini del gioco e non come semplice elemento di contorno, potrebbe rivelarsi una sfida ardua per qualunque sviluppatore.
Esistono fattori comuni che caratterizzano qualunque tipo di game, fattori che ormai da tempo riconosciuti ed attorno ai quali è stata definita una vera e propria *teoria del gaming*. Dal momento che ciò che si dovrà andare a sviluppare sarà a tutti gli effetti un game, sarà dunque quanto meno apprezzabile andare ad analizzarli, ovviamente nella maniera più riassuntiva possibile, dal momento che è sicuramente non necessario chiarire cosa sia esattamente un videogame.

1 Elementi base di un videogame

Così come si può ben intuire dal nome stesso, un qualunque videogioco è composto essenzialmente da due componenti fondamentali: il termine **video**

è relativo alla parte di visualizzazione del gioco stesso, in pratica in tutto ciò che si vede su schermo, mentre il termine **game** fa invece riferimento alla caratteristica di giocabilità del prodotto, ossia alla così detta componente di *game-play*. Il game-play comprende a sua volta tutte le varie forme d'interazione con l'utente, le quali rappresentano sostanzialmente la forma d'input principale per il gioco, ed il modo in cui tutti gli elementi del gioco evolvono nel tempo, in base a delle proprie regole interne, alla storia delle interazioni stesse con l'utente o di altri fattori esterni.

Risulta chiaro come questa evoluzione debba seguire una determinata logica, logica che deve risultare comprensibile al giocatore. E' inoltre necessario che lo scopo del gioco sia ben definito, bisogna cioè fondare l'intera struttura del game-play su un concetto di tipo vantaggi-svantaggi in modo tale che il giocatore sia più propenso a seguire una determinata linea di azioni.

Sia grafica di gioco che comandi d'input sono strettamente dipendenti dal sistema fisico sul quale il videogioco stesso viene eseguito. Per ottenere determinati effetti di rendering serve infatti il giusto supporto grafico lato hardware. Allo stesso modo le interazione con l'utente variano a seconda dei diversi tipi d'input messi a disposizione. I tradizionali tipi di controlli erano legati all'uso di keyboard, mouse o joypad, unici dispositivi d'input disponibili (fino a relativamente poco tempo fa) su consoles o personal computers, mentre oggi si stanno imponendo anche nuovi tipi d'interazione, per mezzo ad esempio di touch-screens o sensori di movimento.

La logica di gioco tende invece ad astrarre dal particolare hardware a disposizione, anche se il modo in cui il game viene codificato dipende dal particolare linguaggio utilizzato per programmare su quello stesso hardware. Dunque, anche se il particolare supporto fisico non influisce direttamente su cosa sia possibile realizzare, questo influenza sostanzialmente il modo in cui il game viene realizzato.

Oltre alla componente grafica e alla componente del game-play, in un videogioco c'è anche la componente *audio* da considerare. Non si tratta in realtà di una parte strettamente necessaria, dal momento che un game potrebbe venir giocato anche senza alcun effetto sonoro, tuttavia rimane qualcosa che contribuisce all'esperienza ludica, ragion per cui ne viene normalmente considerata parte integrante.

Grafica, gameplay e poi sonoro, sono certamente le componenti principali di un qualunque videogioco e sono state sempre presenti fin dai tempi in cui il primo game è stato creato. Tuttavia esistono anche molti altri possibili aspetti che possono essere considerati nella creazione di un prodotto software di questo tipo, soprattutto in relazione alla particolare tipologia di gioco che si intende realizzare. Si tratta di componenti che sono stati man mano presi in considerazione nel tempo, mentre i videogames stessi stessi si sono evoluti.

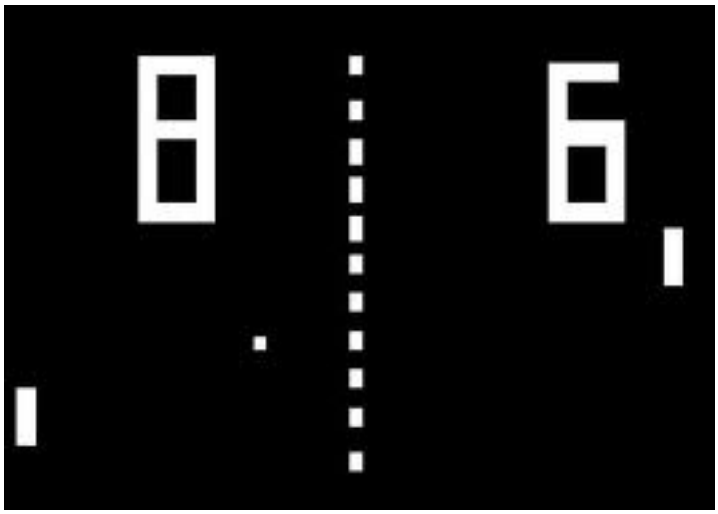

Figura 2.1: Il primo videogame in assoluto, Pong (1972, arcade systems), aveva grafica e comandi essenziali: due pulsanti per barra, per muoverle da una parte o dall'altra

Fra questi aspetti la gestione dell'IA (*intelligenza artificiale*) è certamente stato uno dei primi ad essere considerato, ed in tal senso basti anche solo pensare ai primi giochi di scacchi o di dama, realizzati fin dai tempi dei primi PC, che permettevano già un buon livello di sfida fra il giocatore reale e quello virtuale. E' stata in seguito molto spesso impiegata per gestire il comportamento dei così detti *NPC (Not Player Characters)*, ossia di tutti i personaggi virtuali che non vengono comandati direttamente dai giocatori e che possono aiutare od ostacolare i giocatori stessi durante lo svolgimento del gioco. Deve essere però chiaro che si tratta pur sempre di un comportamento propriamente simulato e fondato in buona parte sull'utilizzo di particolari trucchi o stratagemmi, dal momento che l'implementazione di un vero e proprio processo di ragionamento virtuale rimane ancora oggi qualcosa di estre-

mamente dispendioso in termini di potenza computazionale e poco adatta ad per contesti di real-time.

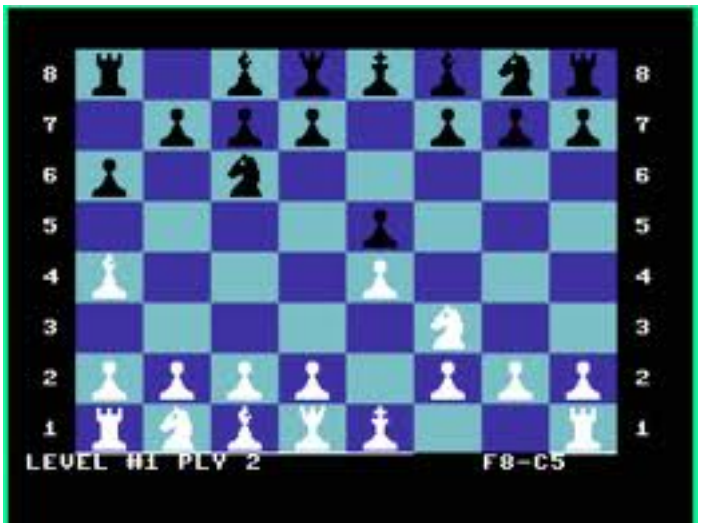

Figura 2.2: Chessmaster (1983, C64) uno dei primi videogiochi di scacchi e già era in grado di offrire un elevato livello di sfida al giocatore

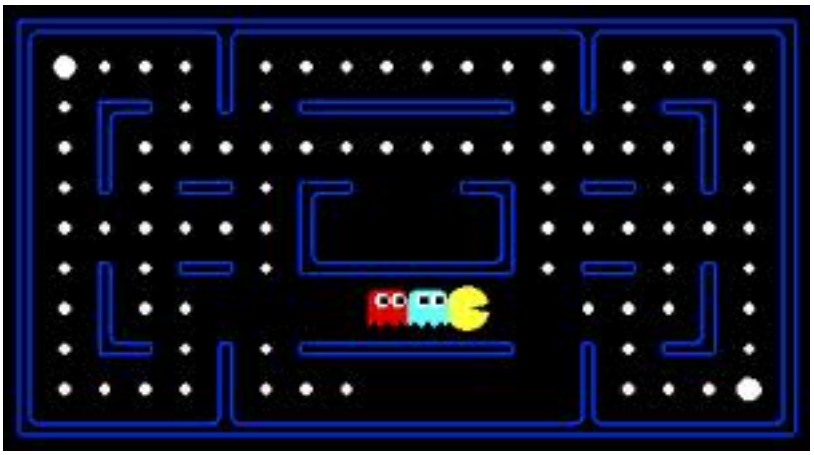

Figura 2.3: Pacman (1980, arcade systems) fu uno dei primi videogame ad applicare rudimentali meccanismi di IA alla gestione del comportamento dei nemici. La scelta della direzione presa da un fantasma ad ogni svolta era influenzata dalla posizione attuale del personaggio di riferimento.

Un altro aspetto è poi la gestione del *multyplaying*, che permette a più di un giocatore di interagire, in maniera cooperativa oppure competitiva, con

lo svolgimento del gioco, ed è dunque legato in particolar modo all'aspetto del game-play. Recentemente è inoltre spesso legato anche alla gestione del *networking*, ossia all'insieme di tutte le funzionalità inerenti all'utilizzo di una qualche tipo di connessione remota, o per permettere l'interazione fra giocatori fisicamente distanti, o per ottenere una qualunque altra informazione esterna riguardante lo stato del gioco. Si tratta in ogni caso di una nuova forma di input nel contesto di un game, rispetto a quella classica tramite strumenti fisici quali keyboard, mouse o controllers di altro tipo da parte di un determinato utente.

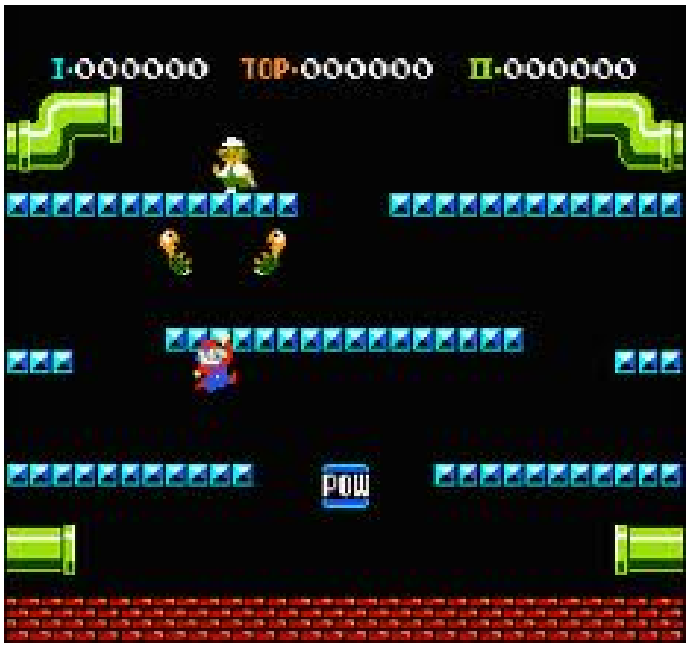

Figura 2.4: Mario Bros (1983, arcade systems) permetteva già un sistema di multiplayng competitivo in locale, tramite una doppia serie di comandi posta in un unico cabinato

Vi è poi la gestione della fisica di gioco, che prevede di simulare un comportamento fisico degli oggetti virtuali come se fossero veri e propri oggetti reali. Si tratta ovviamente di una componente che ha sempre riguardato i videogiochi più complessi, dove si vuole ottenere una simulazione più realistica possibile della realtà, ma ultimamente viene sempre più considerata come un aspetto fondamentale per i games in generale, tant'è che i moderni engine grafici per il supporto al rendering offrono spesso anche funzionalità

Figura 2.5: World of Worldcraft (2005, PC) è invece uno dei più famosi esempi di multiplayer online (in questo caso anzi di *massive multiplaying online*. Permette fino a diverse centinaia di utenti di giocare in un'unica partita.

di motori fisici. In realtà la simulazione della fisica coinvolge anche effetti relativamente semplici quali ad esempio il *collision detection*, per evitare la compenetrazione fra oggetti tridimensionali, attuato quasi sempre per mezzo di *bounding boxex*, ossia di volumi parallelepipeoidali che contengono l'intera geometria del modello. Altri effetti più complicati invece sono ad esempio la gestione della cinematica dei corpi solidi, la gestione della fluidodinamica dei liquidi, la gestione elastica di urti, gli effetti particellari, etc., e ciascuno di questi viene simulato utilizzando metodi differenti.
Per concludere, pur non costituendo questi veri e propri componenti, possono essere considerati come aspetti di cui tener conto nella realizzazione di un videogioco anche il *design* degli elementi del gioco e la stesura di una *trama* (ove possibile), dal momento che sono certamente fattori che ne influenzano la desiderabilità da parte degli utenti.

Figura 2.6: Esempio di gestione di avanzata gestione della fisica (collisioni, esplosioni ed effetti particellari) in un moderno game (Crysis, 2009, PC)

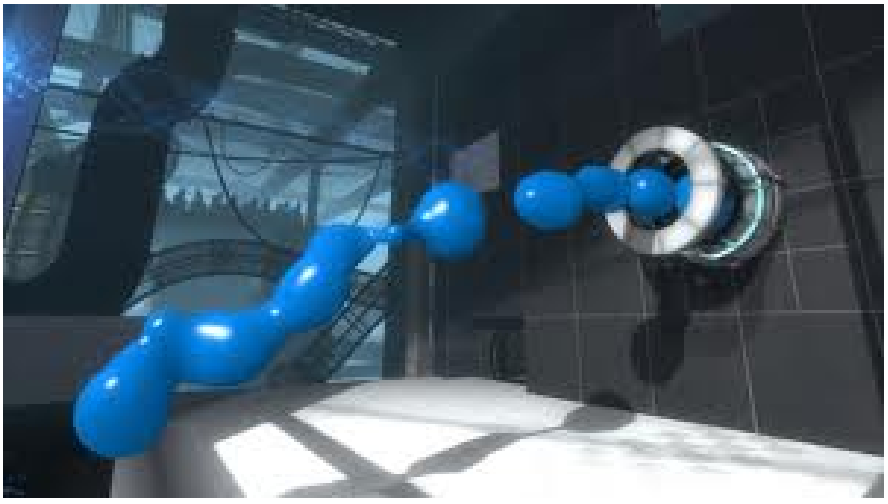

Figura 2.7: Esempio di gestione della fluidodinamica in un moderno game (Portal2, 2011, PC)

Figura 2.8: La sequenza di gioco ripresa nell'immagine, tratta dal videogioco Skyrim (2011, PC), è un perfetto esempio di come nella qualità dell'effetto visivo finale non contribuisca solamente potenza di rendering, ma anche la qualità del lavoro artistico

Capitolo 3

Android e sistemi mobile

1 Sistemi mobile

Uno dei settori tecnologici che ha avuto maggiore sviluppo negli ultimi anni è senza dubbio quello dei dispositivi portatili, ossia tutta quella categoria che va dai semplici telefonini ai più sofisticati smarphones, fino ai moderni tablet. Il salto tecnologico rispetto anche a soli pochi anni fa è notevole, tanto che si può affermare che ormai è cambiato anche il modo in cui questi vengono concepiti. E' passato il periodo in cui un telefonino veniva per lo più considerato come un semplice strumento per effettuare chiamate: ormai anche un comune smartphone di fascia media è dotato di tutta una serie di funzionalità quali GPS, videocamera e registrazione audio, connessione ad internet, memoria di massa per foto, video o audio, e molte altre ancora che lo rendono uno strumento utile in molti contesti. Allo stesso modo anche la potenza di calcolo di cui questi dispositivi sono dotati tende ad essere sempre maggiore e permette il funzionamento di programmi sempre più sofisticati (anche graficamente) tanto da accomunarli sempre di più a veri propri computer e console.
Dispositivi tanto sofisticati e multifunzionali richiedono anche un'adeguata gestione lato software, a partire da veri e propri sistemi operativi per dispositivi portatili. Attualmente ne esistono alcuni esempi, da **Windows Mobile** a **Symbian** per citarne alcuni, ciascuno differente per caratteristiche e dimensioni, anche a seconda del tipo di device per il quale ciascuno di essi è stato pensato. Due sono però quelli di maggior rilievo fra tutti gli altri e che negli ultimi anni hanno riscosso un successo davvero notevole: **iOS** e **An-**

droid. iOS è il sistema operativo di casa Apple, pensato inizialmente per i suoi dispositivi iPod più sofisticati, quindi portato anche sugli smartphone iPhone e per ultimo anche sui tablet iPad; tali dispositivi ne godono ovviamente l'utilizzo esclusivo. Proprio grazie al successo di questi iOS si è da subito imposto come sistema operativo mobile più venduto.

Android è invece il sistema operativo sviluppato da Google, non per un particolare device (a differenza di Apple, Google non produce hardware) ma per qualunque generico tipo di smarphone, tablet o simili, e come iOS si presenta come un sistema molto ben strutturato e performante, capace di gestire molteplici funzionalità a vari livelli, ponendosi così come diretto competitore del prodotto Apple. A differenza di iOS però, Android si basa su principi completamente diversi, se non agli antipodi: mentre il primo ben riflette la politica riservata e chiusa di Apple nel voler custodire ogni tipo di informazione sensibile sui propri prodotti, il sistema operativo di Google è *open-source* ed è stato pensato fin dall'inizio ai fini di portabilità. Molto differenti sono di conseguenza anche i kit di sviluppo, basato su un framework ed un linguaggio proprietario l'uno (Objective-C), utilizzabile con tecnologie e linguaggi (Java e C) completamente liberi e gratuiti, l'altro. Ciò non favorisce di certo la portabilità di una qualunque applicazione da una piattaforma all'altra. Nonostante questo la grande diffusione ormai raggiunta, sia dai dispositivi iPhone ed iPad che da quelli che dispongono di Android, ha spinto sempre più sviluppatori a sviluppare sia per l'uno che per l'altro sistema, adottando il più possibile un approccio *cross-platform*. D'altronde, nonostante le molteplici differenze, molti sono anche i punti in comune e le funzionalità supportate.

Per quanto riguarda il software di supporto che si può avere nello sviluppo di un'applicazione utilizzante realtà aumentata ad esempio, sia iOS che Android possono ottenere una completa e personalizzata gestione della camera e del relativo buffer dell'immagine da utilizzare, così come qualunque altro componente fisico o software di supporto (tipo GPS, sistema di geo-localizzazione o bussola), semplicemente attraverso le relative chiamate API. Inoltre, oltre al loro supporto software di base per la creazione di finestre, pannelli di messaggio o figure bidimensionali, essi consentono entrambi il pieno supporto alle librerie grafiche OpenGL (versione ES, per dispositivi portatili), per

realizzare grafica in 3D.
Dal momento che l'oggetto di studio della presente tesi è il sistema operativo Android, nel seguito si descriverà questo in maggior dettaglio.

Figura 3.1: alcuni esempi di dispositivi mobile di nuova generazione: da sinistra a destra iPad, iPhone, tablet Android e smartphone Android

2 OS Android

Android è il sistema operativo di Google pensato appositamente per dispositivi portatili (per smartphones inizialmente, poi portato anche su tablet), come diretto competitore dell'iOS e quindi dei vari iPhone e iPad di casa Apple, ed è fortemente incentrato sul paradigma open-source, il quale ha sicuramente influito alla sua forte diffusione. In virtù di questa sua caratteristica qualunque produttore hardware può decidere, se lo ritiene necessario, di analizzarne e studiarne il codice, quindi di adattarlo in modo da poterlo far funzionare in modo ottimale sui propri dispositivi, una volta ottenute le licenze da parte di Google, inoltre, qualunque community che decidesse di farlo, può contribuire al miglioramento del sistema stesso, tramite eventuali suggerimenti o proposte. La maggior parte dei grandi produttori di smartphone e tablet (ad eccezione ovviamente di Apple), quali Samsung, Logitec, Sony, etc., ha già adottato Android come sistema operativo di riferimento per molti dei loro device, e altri ancora sicuramente seguiranno questa tendenza.

Figura 3.2: il logo standard di Android

Android si presta dunque molto bene a venir installato su una gamma molto ampia di dispositivi portatili, a prescindere da una qualunque architettura hardware in particolare. Oltretutto neanche Google non ha mai dettato o suggerito degli standard in merito. Per quanto questo aspetto abbia contribuito certamente alla grande diffusione del sistema operativo, c'è però un rovescio della medaglia, poiché proprio la mancanza di un modello di riferimento rende assai difficile prevedere il funzionamento esatto e le prestazioni di un'applicazione, quale ad esempio un game in 3D, su un qualunque device, in particolare se questa va ad utilizzare in maniera intensiva le risorse a disposizione. Questo aspetto può essere un problema per gli sviluppatori, i quali devono testare i propri prodotti su più dispositivi disponibili, scelta però non sempre praticabile.

Il cuore del sistema Android è costituito principalmente dal *kernel Linux* e dalla **Dalvkin Virtual Machine**, una particolare *Java virtual machine* riveduta ed ottimizzata. Java è infatti il linguaggio di programmazione base per creare programmi Android, anche se a partire dalla versione 1.6 è stata aggiunta la possibilità di scrivere applicativi almeno parzialmente in codice nativo (C e C++), tramite un kit di sviluppo ausiliario all'SDK base di Android, detto **NDK** (*Native Development Kit* appunto). A partire inoltre

dalla versione 2.3 è diventato anche possibile scriverli completamente in tal modo. Con il rilascio di ciascuna versione vengono introdotte nuove caratteristiche e funzionalità che contribuiscono ad espandere le potenzialità di Android. Ad ogni nuovo rilascio ufficiale viene associato un così detto **livello di API Android** e viene elencato l'insieme delle nuove funzionalità *stabili* (o *fronzen*, usando il gergo degli sviluppatori) introdotte, ma viene in ogni caso assicurata la retro-compatibilità con le versioni precedenti. Si è detto funzionalità stabili perché con ogni nuova release eventualmente possono venir aggiunte anche delle funzionalità che ancora stabili non sono, in attesa di venir migliorate o completamente scartate con rilasci successivi; è ovviamente sconsigliato il loro utilizzo sebbene possibile, almeno per applicazioni commerciali.

In ogni caso il punto d'ingresso di qualunque applicazione Android rimane il concetto di **activity** (attività), sia che esso sia implementato attraverso una classe Java, sia che venga espressa mediante una struttura in C. Per chiarire che cos'è un'activity potremmo paragonarla all'idea di una schermata Android, dotata di un proprio ciclo vitale, in virtù del quale può continuare ad esistere anche quando non visibile su schermo. Non va però confusa con l'applicazione stessa, dal momento che un'applicazione può essere composta da diverse activity che si succedono sullo schermo secondo diverse modalità (in base ai meccanismi del sistema), ne tanto meno con il concetto di processo o di thread. E' semplicemente il concetto adottato da Android ai fini di un'ottimale gestione delle proprie risorse. Lo sviluppo di un programma richiede dunque la definizione di una o più particolare activity, con tutti i metodi associati, e la dichiarazione di questa come attività da visualizzare ad applicazione iniziata.

Alla semplice stesura del codice, è affiancata anche una programmazione in stile *dichiarativo*, espressa attraverso l'uso di vari file in stile XML, tramite i quali è possibile definire le varie proprietà dell'applicazione che fanno parte dell'insieme dei file risorse delle stessa. Il più importante fra questi è senza dubbio il file **AndroidManifest.xml**, vero e proprio punto d'ingresso dell'applicazione, che contiene, fra le diverse cose, la dichiarazione del livello di API utilizzate, le attività che compongono l'applicazione, compresa la dichiarazione dell'attività principale, ed i vari *permessi* per l'utilizzo di funzionalità

particolari. I permessi (**permissions**) sono uno dei meccanismi di Android ideati ai fini di sicurezza, ed essenzialmente avvertono l'utente qualora un'applicazione che intende installare utilizza funzionalità che si potrebbe ritenere critiche, o perché richiedono un elevato consumo della batteria, o perché eseguono operazioni che potrebbero mettere a rischio la riservatezza dei propri dati, quali ad esempio le chiamate o l'accesso via internet. Altri file di risorse servono invece a definire proprietà quali il layout visivo della finestra entro cui viene eseguita l'activity, il font o il tipo di lingua utilizzati, oppure vengono caricate ed utilizzate a run-time dall'applicazione, secondo diversi metodi. Riguardo quest'ultimo tipo di risorse potremmo dividerli ancora in due categorie: quelle per le quali viene effettuata una compressione a tempo di compilazione, dette **raw**, e quelle invece caricate senza modifica, gli **assets**, direttamente visibili all'interno del file **.apk** finale.

Il file con estensione .apk è appunto il risultato finale nel processo di compilazione-linking di un programma Android, ed è quello che va direttamente trasferito sul dispositivo per installare il programma stesso. Potremmo definirlo il file binario dell'applicazione, anche se in realtà si tratta di un particolar tipo di archivio compresso, simile ai file .zip, ed il processo per la sua generazione richiede diversi passi e l'esecuzione di determinati comandi, ai quali corrispondono le diverse fasi del processo stesso. Non ci si dilungherà nei comandi che servono alla compilazione ed installazione di un programma Android, si ricordi solo che sono tutti compresi fra i diversi tools forniti con l'SDK di Android. Ovviamente per utilizzare le funzionalità delle API più avanzate è necessario scaricarsi sempre l'ultima versione dell'SDK stesso. Nel caso l'applicazione sia parzialmente scritta in nativo, oltre all'SDK è necessario anche l'NDK, il quale si occupa di tradurre i sorgenti scritti in C e C++, nelle librerie proprie di Android, statiche (estensione **.a**) o dinamiche (estensione **.so**). Per far ciò è necessario che tali sorgenti siano contenuti all'interno di una cartella di nome **jni**, assieme ad un file di configurazione detto **Android.mk**, che contiene le informazioni di creazione in un determinato formato, più eventualmente un file di supporto detto **Application.mk**. Anche di questi ci occuperemo in seguito, in particolare in Appendice D, per ora annotiamo solo che la generazione dei file di libreria va fatto prima di quella del file .apk.

Senza dilungarsi ulteriormente sugli altri aspetti dettagliati di Android, che

richiederebbe sicuramente una trattazione molto più approfondita e non sarebbe certo competenza del presente lavoro, andiamo ora ad analizzare quelle caratteristiche del sistema che ci interessano per lo sviluppo della nostra applicazione, ossia quelle contribuiscono all'implementazione della realtà aumentata.

2.1 Supporto al gaming e all'AR in Android

Anzitutto Android permette la completa gestione del dispositivo della videocamera (o dei dispositivi, nel caso ve ne fossero più di uno), con tutte le funzionalità ad essa collegate quali attivazione e disattivazione, regolazione del focus, della luminosità o della risoluzione, ratio di cattura in caso di ripresa video, gestione del buffer d'immagine, il tutto tramite le opportune API. Questo è senza dubbio il requisito principale per la gestione dell'AR. Inoltre sono a completa gestione delle applicazioni che intendessero farne uso anche gli altri componenti fisici, quali GPS o sensore di movimento, e tutti i possibili servizi software d'utilità, come ad esempio il sistema di geo-localizzazione, anche questi tramite le opportune API. Le funzionalità supportate e gestibili sono molteplici, sempre che il dispositivo in questione ne sia fisicamente provvisto. Tuttavia c'è anche da considerare che per la maggior parte queste API sono implementate in maniera stabile solamente in Java, mentre le relative implementazioni in nativo sono per lo più ancora instabili.
C'è inoltre l'aspetto di rendering grafico da considerare nell'AR, ossia in che modo le informazioni aggiuntive vengano virtualmente visualizzate. Oltre ai metodi per la creazione dei propri elementi grafici, quali view, linee di testo o pulsanti, e per il disegno di figure bidimensionali, Android supporta anche la librerie grafiche OpenGL per la grafica 3D, molto utilizzate per esempio nella creazione di videogiochi, nella loro versione ES ossia rivedute ed adattate per il contesto dei dispositivi portatili. Questa è una caratteristica che conferisce grandi potenzialità al sistema operativo, al pari dei sistemi iOS, poiché rende possibile lo sviluppo di applicazioni anche particolarmente complesse dal punto di vista grafico. Inoltre il supporto alle librerie grafiche viene garantito anche in ambiente nativo, e certamente questo è un notevole vantaggio dal momento che garantisce prestazioni certamente maggiori rispetto a quanto si avrebbe in Java. C'è da considerare però che delle tre versioni fin'ora esi-

stenti delle OpenGL ES, la 2.0 (cioè quella che permette l'utilizzo di una pipeline grafica programmabile) è presente solo a partire dalla versione 2.0 di Android, il che pone già dei limiti su quali dispositivi sia possibile realizzare determinate applicazioni.
Le potenzialità offerte dalle librerie grafiche OpenGL ES, combinata con la potenza di calcolo offerta da molti dei moderni dispositivi sui quali ad Android è installato, hanno spesso permesso di raggiungere risultati davvero notevoli, paragonabili a quelli dei più avanzati dispositivi iOS. Esempi sono i game illustrati in Fig. 3.3 ed in Fig. 3.4.

Figura 3.3: Nova 2, un videogioco d'azione in prima persona sviluppato da Gameloft ed uno dei migliori esempi di grafica avanzata su Android

Inoltre OpenGL ES si integra con il sistema di *windowing* (utilizzo delle finestre) reale del sistema operativo, per Android così come per gli altri dispositivi mobile, tramite la libreria EGL, che viene utilizzata per definire il contesto in cui utilizzare le funzionalità di grafica. Tale libreria non è però ancora implementata stabilmente in nativo, per cui devono essere adottati metodi particolari per il suo utilizzo.
Per un ulteriore esempio riguardo le potenzialità messe a disposizione di un dispositivo Android, in Appendice E viene riportato un precedente progetto riguardante lo sviluppo ed il test di un programma di grafica in grado di fungere da benchmark.

Figura 3.4: Order and Chaos, probabilmente il migliore esempio di videogioco mmo (massime multiplayer online) in 3D disponibile su Android

Capitolo 4

Definizione ed analisi dei requisiti di ARGame

Dopo aver analizzato le varie tematiche inerenti al progetto, cominciamo ora ad occuparci dello sviluppo dell'applicativo che dovrà essere realizzato. Per evitare di continuare comunque a riferirlo in maniera del tutto generica, come *applicazione* o simili, è stato scelto il nome **ARGame** per poterlo identificare. Si tratta ovviamente di un nome relativo al contesto di questa tesi e non avrà a che fare con il nome finale del videogioco vero e proprio una volta che sarà realizzato.
La prima fase di un progetto qualunque è la definizione delle funzionalità delle quali un sistema software dovrà disporre e dei requisiti che dovrà soddisfare, una volta implementato.

1 Requisiti applicativi

Fino ad ora è stato solamente detto che ARGame dovrà essere un'applicazione di tipo videogame per Piattaforma Android e facente uso della realtà aumentata. Entrando ora più nei dettagli diciamo che si tratta di un vero e proprio gioco in tre dimensioni (con tutte le conseguenze progettuali che cioè a sua volta comporta) che sfrutta la tecnica dell'AR con utilizzo di markers, quindi che fa uso dei marker stessi per definire posizione e tipo di oggetti virtuali. L'utilizzo di markers rimarrà tuttavia assai limitato poiché, per lo specifico gioco, sarà necessario un solo marker per definire la posizione di

tutti gli oggetti in scena. Il gioco stesso è in realtà assai semplice: sostanzialmente si tratta un puzzle-game che prevede il posizionamento di alcune sequenze di cubi che cadono dall'alto sopra alle corrispettive caselle di una griglia (che deve essere posizionata in corrispondenza del marker stesso), in uno stile simile a quello del gioco del Tetris. Degli aspetti del game-play comunque, ci si occuperà solo in maniera marginale nel contesto di questa tesi.

La versione di Android da considerare come target è la 2.3 (denominata **Gingerbread**), quindi l'applicazione dovrà per forza essere compatibile con qualunque dispositivo che utilizza questa versione o una superiore, anche se viene lasciata la possibilità di renderla compatibile con versioni inferiori. Inoltre viene anche indicato come particolare device da prendere come riferimento il modello Sony Ericsson Xperia Play, per quale si intende ottenere una particolare ottimizzazione. Fra le varie particolarità di questo smartphone, c'è anche la presenza di particolari comandi fisici in forma di slide-pad, che risulterebbero molto interessanti da utilizzare ai fini del gioco. Dettagli sul dispositivo specifico sono forniti in Appendice A.

2 Requisiti non funzionali

Pur non essendo un aspetto esplicitamente richiesto, è chiaro che, in quanto a videogame e quindi per sua stessa natura, l'applicazione dovrà risultare particolarmente reattiva e performante ed evitare il piè possibile cali di prestazioni, sia in termini di frame rate che di interattività con i controlli e l'interfaccia utente, per non minarne la giocabili tè Sicuramente le performance dipendono molto anche dalla risoluzione dell'immagine catturata e riproposta su schermo come sfondo della scena in 3D, e sebbene si potrebbe semplicemente scegliere quella minima è anche preferibile lasciarne l'impatto visivo gradevole entro un certo margine, trovando il giusto compromesso.

E' altrettanto desiderabile avere un efficiente sistema di riconoscimento dei marker, in modo tale che questi vengano identificati prontamente dal sistema una volta inquadrati, ed evitare agli utenti ad esempio di riposizionare piè e piè volte la camera.

3 Ulteriori requisiti progettuali

Quelli appena elencati sono i requisiti che l'applicazione dovrà garantire una volta portata sul dispositivo fisico, tuttavia, ai fini del progetto, ci sono altri aspetti per i quali si intende porre particolare attenzione ed è quindi bene che siano messi bene in evidenza prima di procedere con la parte di progettazione.
Prima di tutto, nonostante l'architettura di riferimento è quella di Android ed il *deployment* finale dovrà avvenire su un dispositivo di questo tipo, si intende adottare il piè possibile un approccio di programmazione di tipo *cross-platform*, affinché in futuro l'applicazione possa essere portata facilmente e senza troppe modifiche anche su altre piattaforme (ad esempio iOS). Per quanto riguarda ottimizzazioni e performance comunque, si può continuare a considerare esclusivamente il device nominato in precedenza.
In secondo luogo si vuole mettere bene in evidenza la parte di sviluppo *del sistema di AR*, ossia tutta quella parte che si occupa dell'implementazione della tecnica della realtà aumentata in sé per sé, dal riconoscimento del marker alla gestione della grafica e degli elementi tridimensionali in scena, e cercarla di distinguerla nettamente da quella di implementazione del gioco vero e proprio. Fra le due è infatti sicuramente la prima quella che maggiormente interessa per questo progetto.
Dai due punti appena elencati si evince (e questo possiamo considerarlo un terzo requisito) che dovrà anche esserci una certa strutturazione, sia a livello di progettazione che a livello di codice, delle varie parti che compongono l'applicazione stessa.

4 Analisi dei requisiti

Prima di procedere con la progettazione, facciamo una breve analisi sui requisiti che sono stati elencati, almeno per quel che riguarda gli aspetti piè rilevanti.
Innanzitutto è stato determinato che ARGame dovrà girare su dispositivi Android di versione 2.3.3 o superiore, quindi le API che dovranno essere utilizzate sono quelle di livello 10 (per ciascun livello si fa riferimento alle note ufficiali, per maggiori dettagli); inoltre fra funzionalità particolari che do-

vranno essere utilizzate, oltre che dichiarate a livello di **manifest Android** come *permissions* ed *uses-features*, vi sono l'accesso alla telecamera e l'utilizzo delle librerie grafiche OpenGL ES per il rendering in 3D. Riguardo a queste ultime, dal momento che ne esistono due versioni, una per la pipeline fissa e una per quella programmabile, come accennato in precedenza, è necessario decidere quale utilizzare fra le due, dal momento che i due tipi sono incompatibili fra loro e l'utilizzo combinato non è possibile (a meno di rischi di errori durante il rendering). La scelta è alla fine ricaduta sull'utilizzo della pipeline programmabile, ma delle relative motivazioni ci si occuperà in fase d'analisi.
La necessita di cercare di mantenere quanto piè possibile un approccio platform-independent, suggerisce di cercare di limitare le parti di codice contenenti le chiamate a funzione proprie di Android e a cercare di tenerle fisicamente distinte da quelle platform-dependet, ed influisce sulla scelta del linguaggio di programmazione da utilizzare, dal momento che, utilizzando le API di livello 10, è possibile anche scrivere interamente l'applicazione in nativo. L'adozione del linguaggio C/C++ permette sicuramente performance migliori, e anche per questo è preferita ove possibile, ma non è detto che sia sempre utilizzabile poiché, come già detto, molte funzionalità di Android necessarie potrebbero non essere nativamente disponibili.
Il corretto funzionamento del sistema di riconoscimento dei markers, per concludere, è una parte particolarmente critica del progetto, per via dell'importanza che riveste nell'intero sistema e della difficoltà nella sua implementazione. Dal momento che potrebbe richiedere un dispendio di risorse non indifferente, l'utilizzo di una libreria o di una qualunque porzione di codice fornita da terze parti potrebbe essere un'eventualità da considerare seriamente.

Capitolo 5

Analisi del progetto di ARGame

Dopo aver elencato ed analizzato attentamente i requisiti applicativi e di progetto passiamo alla fase di analisi vera e propria, i cui risultati ci torneranno estremamente utili nelle fasi successive. Per prima cosa cercheremo di individuare ed analizzare quelle che possono essere le maggiori problematiche da affrontare nell'intero corso di progettazione e sviluppo, ed in seguito di abbozzare l'architettura del sistema in esame nelle sue componenti fondamentali.

1 Analisi del problema

In fase di analisi del problema occorre prima di tutto considerare come poter implementare il complesso sistema di AR, ed in particolare occorre decidere se adottare (almeno in parte) librerie o codice di terze parti, oppure realizzarlo interamente senza tale ausilio in ogni sua parte, dall'elaborazione dell'immagine da camera, agli algoritmi di riconoscimento dei marker, alla gestione di oggetti 3D sovrapposti all'immagine reale, etc. Sicuramente quest'ultima soluzione potrebbe risultare estremamente gravosa, poiché richiederebbe anche un sostanziale approfondimento sulle tematiche toccate, le quali andrebbero a spaziare dai campi della computer grafica a quelli della realtà virtuale (e tutti quelli derivati), oltre che probabilmente comportare un dispendio eccessivo di risorse e risultati non del tutto soddisfacenti. D'al-

tra parte potrebbe essere difficile reperire librerie *free* od *open-source* o parti di codice adatte allo scopo, in particolare per una piattaforma ancora giovane quale Android. Volendo si potrebbe anche pensare di utilizzare una libreria disponibile per un sistema differente ed effettuarne un vero e proprio lavoro di *porting*, modificandola direttamente anche a livello codice, per adattarla al funzionamento in ambiente Android, tuttavia anche in questo caso potrebbe rivelarsi un lavoro eccessivamente oneroso e probabilmente non conveniente.

Tornando alla prima ipotesi, la realizzazione della funzionalità di rilevamento dei markers è in particolare quella che richiederebbe un maggiore approfondimento teorico, in questo caso in materia di *pattern recognition*, e per questo rimane la parte per cui ci sarà sicuramente la maggiore necessità di utilizzare una libreria apposita (sempre che ne possa essere individuata una), ma è possibile pensare all'utilizzo di un supporto esterno anche per le altre funzionalità richieste all'applicazione, come ad esempio la gestione della grafica di gioco. D'altronde si tratta pur sempre di un game in tre dimensioni e per questo potremmo ad esempio pensare all'utilizzo di un vero e proprio motore grafico, eventualmente designato proprio per la realizzazione di videogiochi, il quale potrebbe anche essere usato per gestire altri aspetti del gioco stesso al di fuori della grafica, come per esempio la simulazione *fisica*. In una simile scelta bisogna però considerare diversi aspetti: prima di tutto dovrebbe ovviamente trattarsi di un motore compatibile per la piattaforma Android (requisito probabilmente ancora assai difficile da soddisfare) oppure non troppo difficile da venire convertito, poi dovrebbe anche garantire un buon compromesso fra potenzialità e semplicità, oltre che pesantezza ed impatto sulle performance, quindi dovrebbe anche risultare non difficile da utilizzare nel contesto di questo specifico tipo di applicazione, cioè dovrebbe permettere senza troppi problemi l'implementazione della tecnica della realtà aumentata.

In ogni caso il vincolo principale per l'utilizzo di una libreria esterna in Android è che questa sia stata scritta usando o il Java o il C/C++, gli unici linguaggi di programmazione utilizzabili. Generalmente, fra i due, il secondo è preferibile poiché di solito permette performance migliori. Utilizzando direttamente le API di livello 10 l'intera applicazione potrebbe anzi essere scritta in nativo ma, come già detto, occorre valutare se tutte le funzionalità

necessarie vengono rese disponibili anche nativamente. Da quanto possiamo direttamente constatare esaminando la documentazione ufficiale, ancora non sono presenti, fra le altre cose, delle API per il supporto stabile alla gestione della telecamera, il che già ci costringe a passare dall'utilizzo di Java almeno per alcune parti. Nel caso in cui si avessero comunque parti di codice scritte in C/C++, l'accesso ai relativi metodi e funzioni dovrà allora avvenire tramite i metodi della **Java Native Interface** (o **JNI**), supportati in ambiente Android.

Dai requisiti sappiamo che il livello di API scelto non deve eccedere il 10, per rendere l'applicazione finale utilizzabile almeno su dispositivi con versione 2.3.3 di Android, ma questo non ci impedisce l'uso di un livello inferiore, in modo da renderla compatibile anche con dispositivi aventi una versione inferiore, sempre che questo sia quanto meno sufficiente per garantirci il supporto a tutte quelle funzionalità di cui avremo bisogno. Volendo è possibile partire con la fase di progetto prendendo in considerazione un livello di API basilare (ad esempio la versione 1.6.0) e decidere successivamente se incrementarlo a seconda che si necessiti o meno di nuove funzionalità Tuttavia si ritiene anche che già da ora sarebbe bene partire facendo alcune considerazione sulle caratteristiche principali che ciascun livello mette a disposizione e sulla possibilità del suo utilizzo.

Cominciando a fare delle considerazioni proprio con il livello 10, abbiamo già affermato che con questo viene fornita la possibilità di creare un'applicazione completamente in codice C/C++ (funzionalità che però non ci interessa per i motivi già elencati). Ora aggiungiamo che esso fornisce anche una rinnovata gestione di molti sensori, quali ad esempio il giroscopio, e degli *assets*, che ora possono venire utilizzati anche da nativo, in maniera simile a quanto avviene in Java. Questo è un aspetto che può tornare assai utile nel caso di necessario utilizzo di file esterni associati all'applicazione stessa, poiché, sebbene possibile, è sempre preferibile evitare l'accesso diretto al file system di Android tramite i classici meccanismi di input/output, a causa di ben noti problemi di stabilità che questi comportano su questa piattaforma. Ragion per cui, se ci sarà la necessità di utilizzare tali file, si passerà a questo livello di API.

Per quanto riguarda il livello 9 invece, che corrisponde alla versione 2.3.0 di

Android, viene ritenuto praticamente superfluo, poiché qualunque dispositivo con tale versione è aggiornabile direttamente alla 2.3.3, quindi in questo caso viene consigliato direttamente l'utilizzo del livello 10.
Il livello di API 8, corrispondente alla versione 2.2 di Android, si rende assolutamente necessario se si intende utilizzare la libreria grafica OpenGL ES di versione 2.0, poiché è proprio a partire da tale livello che questa viene resa disponibile. Inoltre vengono per la prima volta messe a disposizione anche una serie di nuove funzionalità per una migliore gestione della stessa camera, fra le quali anche la scelta fra la classica modalità *landscape* e la modalità *portrait*, e nuovi modi di utilizzo del buffer immagine. Dal momento che tali funzionalità ci appaiono piuttosto utili ai fini dell'applicazione, poiché ci permettono ad esempio la coordinazione fra la cattura dell'immagine e la sua lettura tramite la gestione del buffer immagine stesso (possiamo impostare la camera in modo tale che ogni volta il capturing venga disabilitato finché il buffer non è stato letto), questo sarà il livello che per ora prenderemo come riferimento.

2 Analisi architetturale

Cerchiamo ora di comprendere ed analizzare quella che può essere l'architettura finale del nostro sistema ARGame, sia dal punto di vista della **struttura** che delle **interazioni** interne, prendendo ogni volta in considerazione le diverse e possibili parti in cui questo si compone e le funzionalità che ciascuna di esse mette a disposizione delle altre. Ovviamente l'architettura complessiva così schematizzata non sarà quella definitiva, ma verrà ulteriormente ampliata e modificata in fase di progetto.

2.1 Struttura del sistema

Come evidenziato in Fig. 5.1, che illustra la struttura stessa del sistema, possiamo per prima cosa fare una distinzione fra quelle parti che dipendono dalla particolare piattaforma in questione, ossia appartenenti all'ambiente Android, e quelle che risultano invece platform-independent. Questo accor-

gimento aiuterà certamente a rendere ARGame maggiormente portabile da un sistema all'altro.

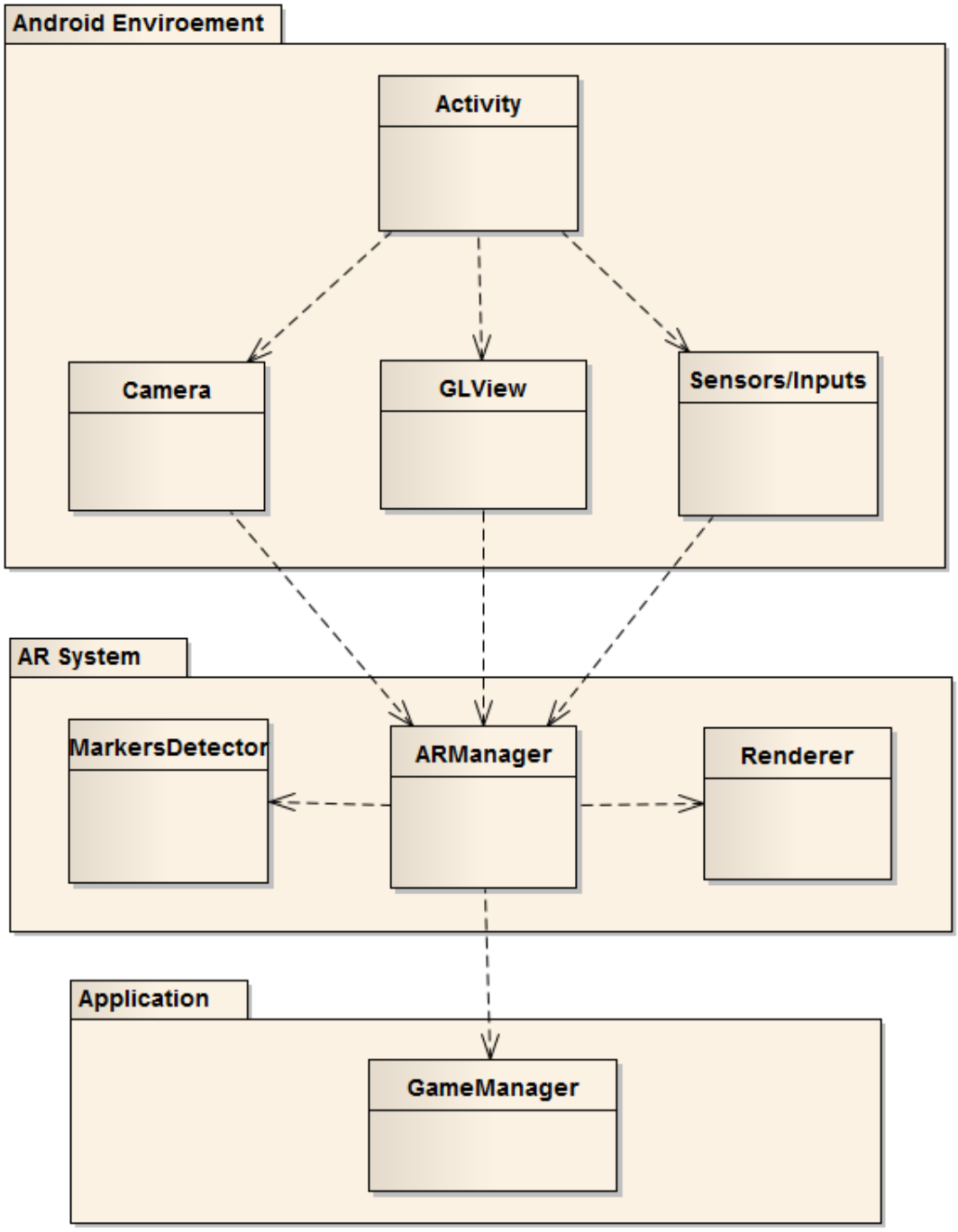

Figura 5.1: analisi: struttura del sistema

Sappiamo che un qualunque programma Android deve implementare almeno un componente fondamentale di tipo *activity*, che rappresenta il vero e proprio punto d'ingresso dell'applicazione e ne stabilisce il ciclo vitale. Inoltre, per applicazioni che fanno uso delle librerie grafiche OpenGL ES per funzioni di rendering in 3D, è anche necessario di un particolare tipo di *view*, denominata **GLView**, che permette la definizione di un opportuno *contesto di rendering* e l'implementazione delle diverse funzioni di *callback*. Oltre a questi due, sempre per quanto riguarda la parte platform-dependent, possiamo anche contare la presenza di un componente appositamente dedicato alla gestione della camera, poiché, come è già stato detto, si tratta di una funzionalità strettamente correlata al sistema operativo in questione.
Possiamo allora considerare tutti i restanti componenti come non dipendenti dallo specifico ambiente di Android. In sostanza si tratta dell'intero sistema che contribuisce a creare l'effetto di AR. Fra questi componenti possiamo certamente individuare quello che serve per effettuare il *marking-detection*, a partire dall'immagine in input ottenuta dalla camera, ed il componente per il rendering grafico. Infine deve esserci anche un componente che faccia uso delle loro funzionalità per garantire l'atteso comportamento del gioco finale. Dal momento che si ritiene però importante soddisfare il requisito di progetto di mantenere ben distinta la parte relativa all'implementazione del sistema di AR da quella propriamente applicativa, è bene considerare ben due componenti distinti a questo scopo. Si ha allora un componente che funge essenzialmente da coordinatore fra riconoscitore e modulo grafico, **AR Manager**, per la realizzazione dell'effetto di realtà aumentata, ed uno che gestisce lo stato del gioco vero e proprio, **Game Manager**.
Da notare come nel modello previsto non vi sia associazione ne fra camera e riconoscitore dei markers, per un passaggio diretto dell'immagine da analizzare, così come non vi è associazione fra la view ed il renderer, in quanto ogni flusso di dato e di controllo, proveniente da parte platform-dependent, passa sempre per il componente ARManager.

2.2 Interazioni fra le parti

Dopo aver identificato la possibile struttura del sistema, evidenziando le parti in cui questo si compone e le funzionalità con quali interagiscono, schematiz-

ziamo piè attentamente il modo in cui tali interazioni si susseguono nel tempo, utilizzando il diagramma apposito illustrato in Fig. 5.2. Vi è ovviamente una corrispondenza diretta con le parti del diagramma precedente. Lo schema è di per sé molto intuitivo ma, senza dilungarci ulteriormente, facciamo solamente notare che per le prime tre azioni (**start view**, **start camera** e **start sensors**) non vi è uno stretto vincolo temporale ma è comunque necessario che queste avvengano prima della successiva azione **update view**.

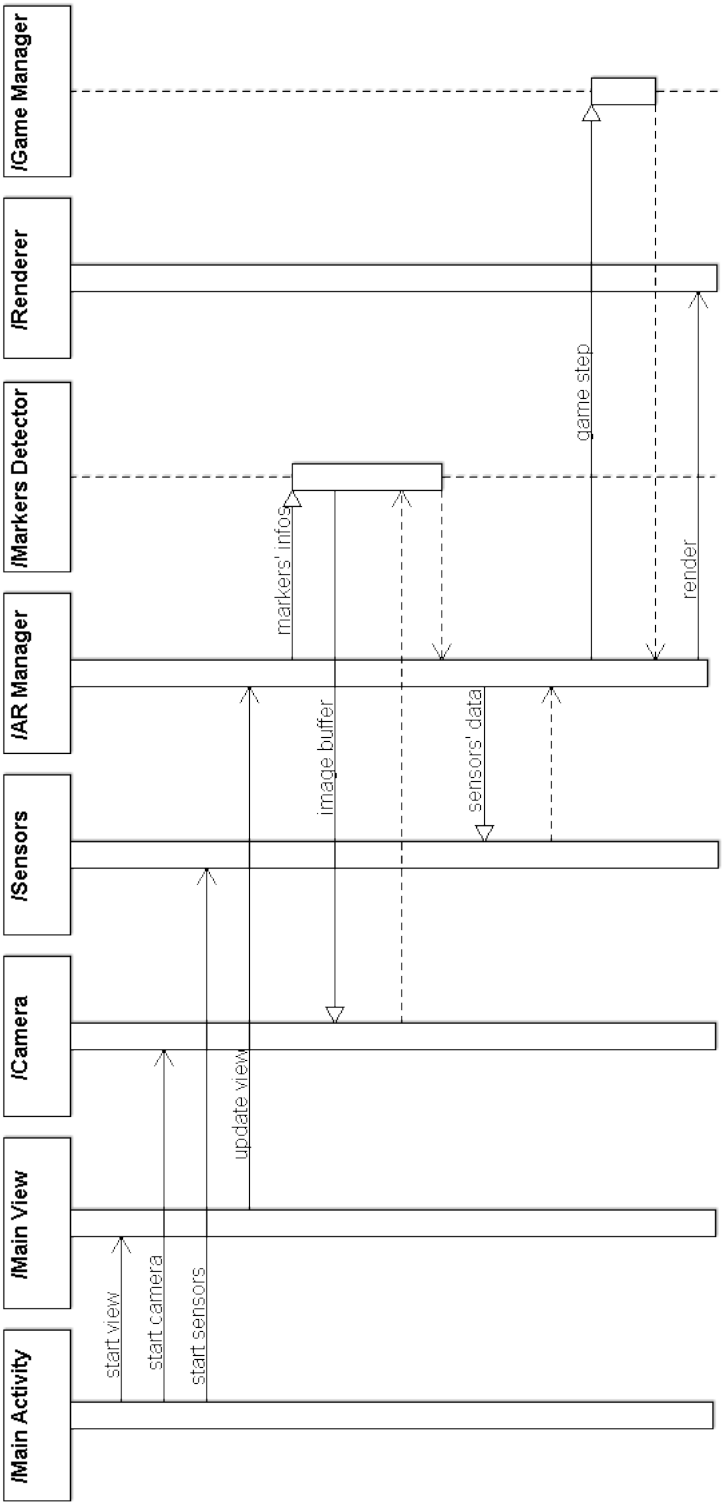

Figura 5.2: analisi: interazioni del sistema

Capitolo 6

Progettazione del sistema di AR di ARGame

Passiamo ora alla fase di progettazione vera e propria del sistema di ARGame. Analizzate quella che può essere l'architettura complessiva e tutte le possibili problematiche da affrontare, bisogna adesso definire le diverse scelte e strategie implementative delle singole parti, che nel complesso contribuiscono a realizzare le funzionalità dell'intera applicazione. Se necessario si andrà anche a rivedere e modificare quanto già individuato in fase d'analisi, eventualmente identificando nuovi componenti.
Per chiarezza questa intera fase è stata divisa un due parti distinte: prima si considererà la realizzazione del sistema di realtà aumentata che funge da fondamenta per il funzionamento di un gioco in AR, quindi quella dell'applicazione game vera e propria. In questo capitolo si approfondiranno i componenti che realizzano le funzionalità del sistema di AR.

1 Modulo di riconoscimento dei markers

La scelta sul come implementare il modulo di riconoscimento dei markers, è ricaduta per l'opzione evidentemente piè conveniente, cioè di utilizzare una libreria esterna che fornisse le funzionalità richieste: si tratta di **ARToolKit**, progettata per l'implementazione di tecniche di realtà aumentata e basata sull'utilizzo di markers. E' di fatto una libreria ben conosciuta e collaudata, in grado di garantire una buona precisione nell'operazione di rilevamento, a

patto di impostare correttamente una serie di parametri di configurazione. Inoltre presenta il vantaggio di essere implementata in C e C++ (a seconda della versione) quindi facilmente integrabile nel nostro contesto.
Utilizzando sofisticati metodi di *pattern recognition*, ARToolKit mette a disposizione diverse funzionalità che permettono di rilevare la presenza dei markers all'interno di un'immagine passata in input. Per ogni marker poi viene ricavata anche una nutrita serie d'informazioni, in base al modo in cui questi appaiono, fra le quali anche le matrici che determinano la trasformazione del marker nell'ideale spazio di vista della camera. I marker di ARToolKit permettono di definire al proprio interno anche dei *template*, particolari forme che vengono riconosciute ed identificate dal sistema in modo da poter effettuare una distinzione fra i markers stessi.
Per ulteriori approfondimenti sulla libreria si faccia riferimento all'appendice B.
L'implementazione che si è deciso di utilizzare per il progetto è quella di di ARToolKit Plus (versione in C++, riveduta e migliorata, della libreria originaria in C) versione 2.2.0, essendo questa anche l'ultima di quelle esistenti. Si tratta però di una versione per piattaforma Windows (apposite per Android non ne esistono), per cui un determinato lavoro porting è stato necessario. Anche se limitate ad un numero ridotto, alcune sostanziali modifiche a riguardo sono state necessarie, poiché alcune delle funzionalità della libreria non risultavano completamente compatibili con la piattaforma. Ad esempio, per effettuare la configurazione di determinati parametri, era previsto anche l'utilizzo di appositi file dati caricati direttamente da file-system, modalità di accesso che però abbiamo già visto non essere del tutto stabile in Android.

2 Gestione della camera e del buffer immagine

Per la gestione della camera e del buffer immagine ottenuto da questa è stato realizzato un modulo apposito, concettualmente separato dal resto del sistema, che permette di gestirli e configurarli in maniera ottimale e trasparente. Nella sua realizzazione è stata posta particolare attenzione a tutti i possi-

bili accorgimenti che contribuivano a renderne ottimali le prestazioni d'uso, aspetto generalmente critico quando si utilizzano componenti particolarmente dispendiosi in termini di risorse come appunto la camera. Una funzionalità certamente molto utile a riguardo è stata ad esempio la possibilità di effettuare una lettura bufferizzata dell'immagine in preview: utilizzando tale opzione è possibile memorizzare ogni immagine ripresa da camera in un apposito buffer e bloccare la ripresa stessa fino a che questo non viene acceduto in lettura. In questo modo è possibile limitare fortemente il consumo e l'uso della CPU.

L'intero modulo va implementato in Java, poiché non esistono API stabili per la gestione della camera in nativo. Il riconoscitore dei markers, al quale il contenuto del buffer d'immagine va passato, è però implementato in C++, ragion per cui il contenuto del buffer stesso va passata al livello sottostante mediante le chiamate a funzione JNI implementate da Android. Nel fare ciò bisogna ovviamente fare attenzione al diverso formato dei dati del C rispetto al Java, per assicurarsi di passare l'informazione in maniera corretta.

3 Conversione del formato immagine

C'è un altro aspetto da considerare, prima di passare i dati dell'immagine alla libreria di ARToolKit e farle eseguire la procedura di pattern-recognition: i formati che di norma vengono accettati dalla libreria sono il *grayscale* (un byte per pixel per determinare la tonalità di grigio) o l'*RGB* (tre byte per pixel per determinare ciascuna componente colorimetrica), mentre quello usato generalmente da una camera digitale è differente. In Android il formato comunemente usato è il *YUV2* ed è ovviamente differente dall'RGB. E' anche l'unico formato per il quale ne è sempre garantito il supporto. Anche se molti device supportano anche l'RGB, il suo utilizzo precluderebbe il funzionamento dell'applicazione su tutti gli altri e questo è senza dubbio un risultato inaccettabile.

L'unica soluzione possibile è quella di ottenere l'immagine nel formato YUV2 ed effettuarne la conversione prima del passaggio ad ARToolKit. Per ovvie ragioni di prestazioni tale conversione verrà effettuata a livello nativo. Una parte apposita per questa funzione è dunque stata realizzata.

4 Modulo grafico

Così come per il riconoscimento dei markers, anche per la gestione della grafica di gioco è stato stabilito di avvalersi di un software di terze parti, piuttosto che utilizzare direttamente le chiamate a funzione OpenGL. Si tratta in questo caso di **Horde3D**, un vero e proprio engine grafico per il rendering 3D basato sull'utilizzo delle OpenGL (versioni desktop), per sistemi *Windows*, *MacOS* e *Linux*, e ottimizzato per realizzare complesse tecniche di rendering. Allo stesso tempo però, nonostante le funzionalità messe a disposizione, risulta anche un motore di semplice utilizzo e non troppo pesante in termini di linee di codice, il che certamente non rende impossibile il suo adattamento per la nuova piattaforma. Anche in questo caso infatti si è dovuto convertire ed utilizzare un software per una piattaforma differente, dal momento che un vero e proprio motore grafico appositamente progettato per Android per il momento non esiste.

A differenza del precedente lavoro di conversione però, il vero problema in questo caso sono state proprio il tipo di librerie grafiche utilizzate dall'engine, ossia le classiche OpenGL per sistemi desktop, quando invece le uniche supportate dal sistema operativo Android sono le OpenGL ES (per sistemi mobile). Nonostante le funzionalità di base siano le medesime, nel complesso queste ultime ne supportano solo un sotto-insieme delle prime, quindi vi è stata una sostanziale incompatibilità da risolvere. Per questo molte delle funzionalità originarie di Horde3D hanno dovuto essere pesantemente modificate o addirittura eliminate. Inoltre anche i file di libreria contenenti parti di codice nel linguaggio di shading GLSL hanno dovuto essere adattati al diverso tipo di linguaggio di shading usato dalle OpenGL ES. Per un esame piè nel dettaglio su tutte le modifiche si rimanda ancora una volta alla relativa appendice C.

Si è appena parlato di file contenenti parti di codice in linguaggio di shading e questo proprio perché Horde3D è un engine grafico fortemente basato sull'utilizzo della pipeline di tipo programmabile e degli shaders stessi, i quali costituiscono vere e proprie risorse del sistema. Questo esclude automaticamente la possibilità di poter utilizzare una versione diversa dalla 2.0 delle OpenGL ES, per il porting su piattaforma Android. Si utilizzerà dunque

questa per il progetto, e la si dichiarerà a livello di *manifest*. L'incompatibilità fra i due tipi di pipeline per quanto riguarda le OpenGL ES avrebbe potuto presentare un notevole inconveniente nel caso l'engine avesse fortemente utilizzato anche delle chiamate a funzione proprie della pipeline fissa, ma fortunatamente il loro uso è assai limitato a poche funzionalità le quali sono state facilmente modificate o eliminate.
In ogni caso, sia per ARToolKit che per Horde3D, c'è la necessità di utilizzare dei particolari file esterni di configurazione, che devono essere dunque memorizzati come *assets*. Dal momento che tornerebbe senz'altro utile che questi file assets venissero caricati direttamente da codice nativo, prendiamo a questo punto del progetto la decisione di utilizzare in definitiva le API di livello 10. Questo è il massimo livello consentito dai requisiti dell'applicazione, come già analizzato in precedenza, per cui non ci saranno altri cambiamenti a riguardo.

5 Rendering del background

Ai fini di riprodurre l'effetto proprio di un'applicazione di tipo AR è necessario anche pensare in che modo renderizzare l'immagine reale, così come viene presa da telecamera, sullo sfondo della scena virtuale. La tecnica usata a questo scopo è in realtà assai semplice e consiste nel porre il contenuto di tale immagine in una apposita texture ed quindi applicare questa ad una superficie visualizzata in proiezione ortografica (anziché prospettica), in modo da riprodurla come sfondo. Si fa però anche in modo che tale superficie non vada a scrivere sullo *Z-buffer*, in modo che tutti gli elementi del game vengano sovrascritti senza problemi sul *color buffer*. Inoltre, anche se non strettamente necessario, è bene utilizzarne una texture quadrata con una misura dei lati che risulta essere una potenza di 2: in considerazione di come funziona il processo di *texturing*, tutto questo permette di ottenere performance migliori. Come conseguenza però, prendendo per ipotesi una superficie geometrica di sfondo che si adatta perfettamente alle dimensioni dello schermo, le coordinate texture della superficie stessa vanno adattate a seconda delle dimensioni dell'immagine passata da camera, per evitare la presenza di spazi bianchi, e allo stesso modo anche la texture va riempita coerentemente.

Per il rendering di questa superficie si potrebbe ancora utilizzare l'engine grafico utilizzato per il rendering di tutti gli elementi virtuali del gioco, ma alla luce degli accorgimenti appena menzionati e del modo in cui i dati dell'immagine dovrebbero essere passati all'engine stesso, si è preferito fare una piccola eccezione allo schema ed utilizzare direttamente le chiamate a funzione OpenGL. Ovviamente, considerando che le API da utilizzare sono quelle di OpenGL ES 2.0, bisogna implementare ciò secondo le modalità di una pipeline di tipo programmabile, ossia tramite l'utilizzo di appositi shaders.

6 Gestione degli input

Dal momento che non vengono utilizzati particolari sensori da utilizzare in combinazione con l'AR, le uniche sorgenti d'input sono costituite dagli eventi relativi a tasti e touchscreen del device, che vengono utilizzati per impartire i diversi comandi di gioco. L'idea per una gestione intelligente degli input stessi, è quella di fornirne una sorta di *snapshot* (per fornire con un'unica informazione molteplici dati) di tutti i tasti premuti e le zone dello schermo toccate in un determinato, ad intervalli regolari fra un rendering e l'altro.

7 Modulo di coordinazione fra marker-detection e rendering della scena virtuale

Rimane ora da comprendere come coordinare le diverse componenti fin qui individuate in modo da ottenere il funzionamento atteso dal sistema. In particolare occorre capire come coordinare le funzionalità di ARToolKit con quelle del motore grafico per il corretto posizionamento dei modelli in scena. A questo scopo, le informazioni essenziali da ricavare riguardano le matrici di trasformazione (di *modelview* e di *proiezione*), che applicate ad un modello permettono di collocarlo esattamente sopra il marker nell'elaborazione finale dell'immagine, e l'identificativo del template, se si vuole associare determinati modelli a determinati markers. Si potrebbe in vero lasciare che sia la parte che realizza il game stesso ad occuparsi di questo compito, ma in questo modo non si otterrebbe di certo un buon livello di modularità Piutto-

sto si è preferito definire un apposito modulo di coordinazione che permetta l'associazione modelli-markers e posizioni i modelli stessi sopra i markers, in maniera trasparente alla logica di gioco. Tutto quello che basta allora fare, da parte applicativa, è la *registrazione* di un determinato modello e dell'ID del corrispondente marker. Quel che effettivamente viene realizzato in questo modo è un livello intermedio di funzionalità fra il game vero e proprio e le librerie Horde3D e ARToolKit.

Per quanto riguarda la procedura appena descritta c'è però da fare una precisazione: Horde3D permette sé di specificare la matrice *world* di qualunque modello in scena, per il suo posizionamento nello spazio, ma non permette in alcun modo di specificare direttamente la matrice di *vista* (o *view*) e la matrice di *proiezione*, che vengono invece calcolate a partire dai parametri del nodo di scena di tipo telecamera utilizzato per il rendering della scena stessa. Per quanto riguarda la matrice di vista si fa semplicemente un'assunzione, ossia che la telecamera stessa sia posizionata nell'origine del sistema di riferimento dello spazio mondo e rivolta in senso entrante rispetto allo schermo di proiezione (asse Z positivo): in questo modo la matrice di vista risulta uguale alla matrice identità e può non venire considerata nel calcolo finale. Si tratta di un'assunzione piuttosto forte nel contesto di un qualunque videogioco, tuttavia è abbastanza coerente con un'applicazione di tipo AR.

Per la matrice di proiezione si è invece deciso di agire a livello di shaders, definendo quattro diverse variabili uniform di tipo *vec4* che rappresentano la matrice stessa decomposta in quattro vettori colonna e vengono inizializzati con i valori della matrice di proiezione ricavata mediante ARToolKit. Queste sono state convenzionalmente nominate come **projection_matrix_1**, **projection_matrix_2**, **projection_matrix_3** e **projection_matrix_4**. Per utilizzare una matrice di proiezione che risulti coerente all'interno di un *vertex shader*, occorre dunque dichiarare e comporre assieme queste quattro variabili. Si tratta in fin dei conti di una soluzione non troppo dispendiosa. L'unica alternativa sarebbe stata quella di modificare la libreria dell'engine grafico anche a livello d'interfaccia, soluzione certamente non molto desiderabile.

Infine questo componente ha anche il compito di caricare e leggere il contenuto dei vari file asset memorizzati dall'applicazione, e di utilizzarli per inizializzare i parametri di configurazione di ARToolKit e le risorse usate da

Horde3D per la gestione degli elementi di gioco. Per semplicità vengono letti tutti i file disponibili all'interno della folder *assets*, seguendo una prefissata struttura delle sotto-cartelle interne.
Riguardo questo aspetto però qualcosa in piè si è voluto fare per i file di configurazione dei markers di ARToolKit. Per il modo in cui funziona l'API stessa della libreria, quando il contenuto di uno di questi viene passato all'apposita funzione, vanno specificati anche altri parametri relativi a ciascun marker come ad esempio le dimensioni reali ed in spazio OpenGL, lo spessore dei bordi o le unità di suddivisione del template interno. Dovrebbe in tal caso essere lo specifico game ad impostare questi valori, sulla base dei markers che si intendono utilizzare. Si Ã¨ invece deciso di optare per un approccio piè di tipo dichiarativo, ponendo nella stessa folder anche un file di tipo testuale contenente i nomi di tutti i file di configurazione dei markers che si intendono caricare, assieme a tutte le altre informazioni appena citate. Si è inoltre fornita la possibilità di specificare l'ID di template che si intende utilizzare per ogni marker: di norma infatti gli ID assegnati da ARToolKit, ad ogni nuovo caricamento di marker, corrispondono a valori in diretta successione, a partire da 0, e la scelta di un determinato identificativo non è normalmente possibile. Il sistema assegna così un secondo ID per la gestione del marker, mantenendo un mapping fra questo e quello originario.

8 Architettura del sistema

Alla luce di quanto fatto possiamo ora definire meglio l'architettura del sistema che abbiamo già schematizzato in fase d'analisi.

8.1 Struttura del sistema

In Fig. 6.1 viene visualizzata la struttura definitiva del sistema.
Lo schema è piè o meno lo stesso di quello già individuato in fase d'analisi, con in piè solamente il modulo per la conversione del formato d'immagine e quello per il rendering del background. Inoltre c'è da notare una rottura per quanto riguarda il funzionamento del sistema fin qui individuato, per quanto riguarda l'interazione fra GameManager e Renderer di gioco, prima

non presente. E' una chiara scelta che è stata fatta per alleggerire il carico implementativo nella realizzazione del game, evitando di passare per il componente ARManager per ogni singola azione di rendering.
Potremmo ora evidenziare come l'architettura dell'intero sistema possa essere inoltre considerata operante su diversi livelli, a partire dal sistema operativo Android stesso, assieme alle proprie librerie e API messe a disposizione, poi con il sistema per la gestione dell'AR fin qui definito, con un proprio livello di funzionalità, ed infine l'applicazione di gioco vera e propria definita su questo. Di questa vista ne abbiamo una schematizzazione in Fig. 6.2

8.2 Interazioni del sistema

Dallo schema delle interazioni in Fig. 6.3 invece, illustriamo come i diversi componenti architetturali appena schematizzati interagiscono in senso propriamente cronologico.
Le azioni **init application**, **init camera**, **init view**, **init game scene** ed **init game**, riguardano l'inizializzazione di singoli o dell'intero sistema, e per tanto vengono eseguite una sola volta; quest'ultima funzione in particolare deve restituire tutti i parametri di gioco necessari ad essere utilizzati dal sistema di gestione AR, quale ad esempio gli ID dei marker utilizzati. Tutte le azioni che invece avvengono dopo riguardano l'intero ciclo di ripresa, avanzamento del gioco e di rendering e quindi vengono ripetute ciclicamente. Il tutto avviene secondo uno schema piuttosto intuitivo fin qui delineato: il componente view, che regola l'intero ciclo di callbacks, richiede dapprima il buffer dell'immagine al componente della telecamera, poi ne passa il contenuto al modulo di coordinazione del sistema di AR, il quale a sua volta ne richiede la conversione di formato per poter effettuare i due passi successivi, ossia di rendering del background e di rilevamento dei markers. Infine viene invocato il comando per il rendering della scena virtuale, che può venir effettuato direttamente dall'**AR Coordinator**, oppure per mezzo del componente **Game Manager** ed in questo caso viene effettuato anche il calcolo del prossimo step nello svolgimento del gioco.

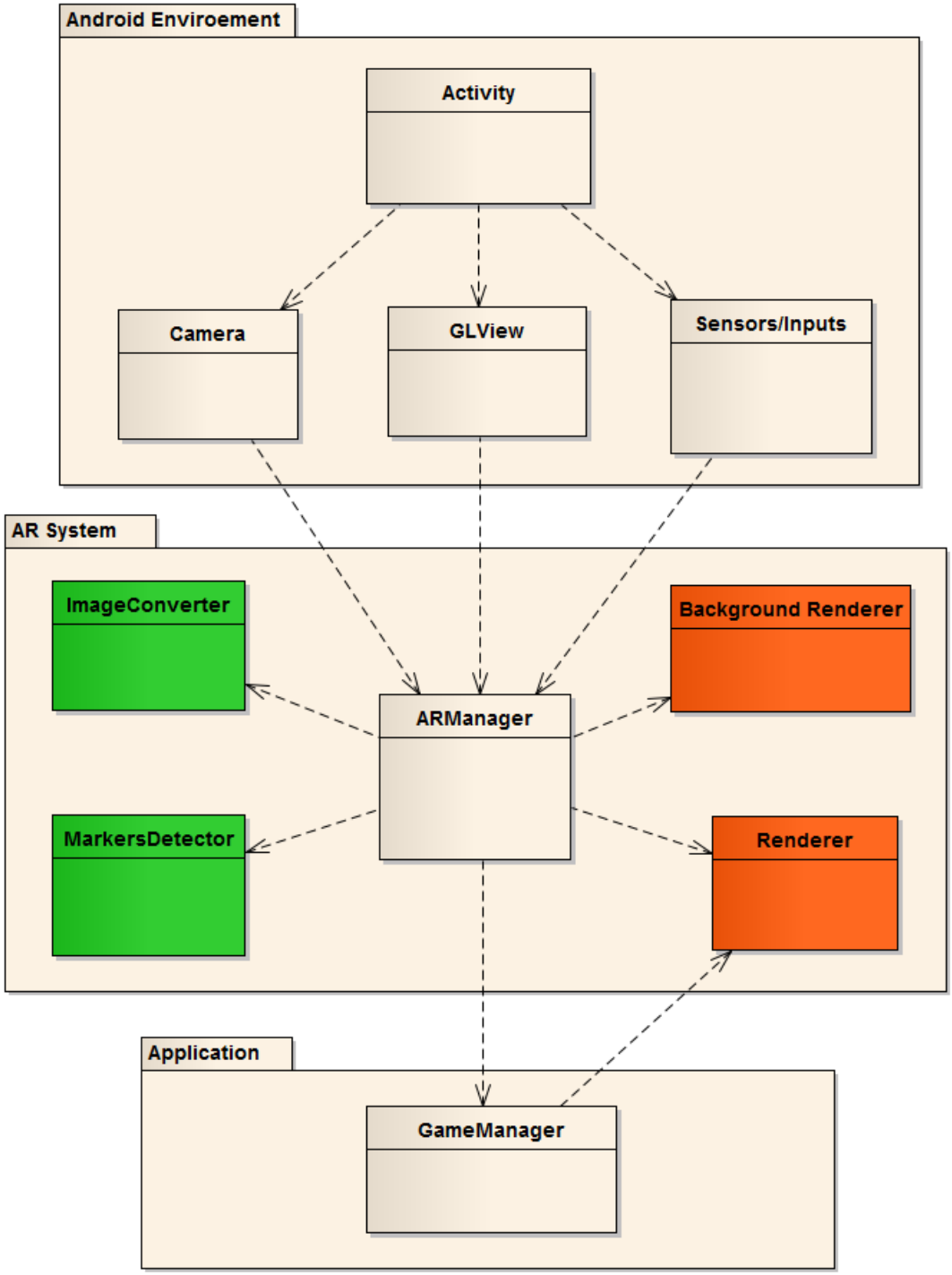

Figura 6.1: progetto: struttura del sistema. All'interno della parte per la gestione del sistema di AR sono state evidenziate in verde le parti relative al blocco CV (compouter vision) ed in rosso quelle relative al blocco CG (computer graphics).

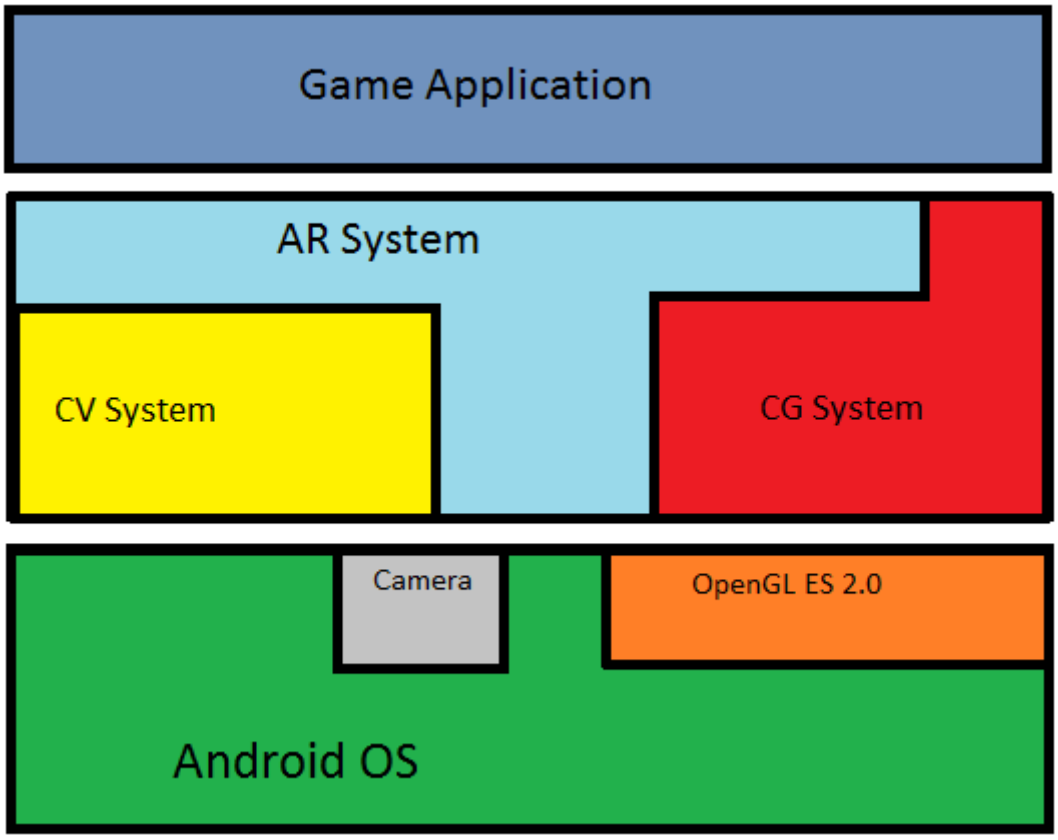

Figura 6.2: progetto: schema architetturale a livelli del sistema di ARGame. Come si può notare le basi sono costituite dal sistema operativo Android stesso, sopra al quale è costruito l'intero sistema di gestione della realtà aumentata. Sopra di questo viene a sua volto realizzato il gioco vero e proprio

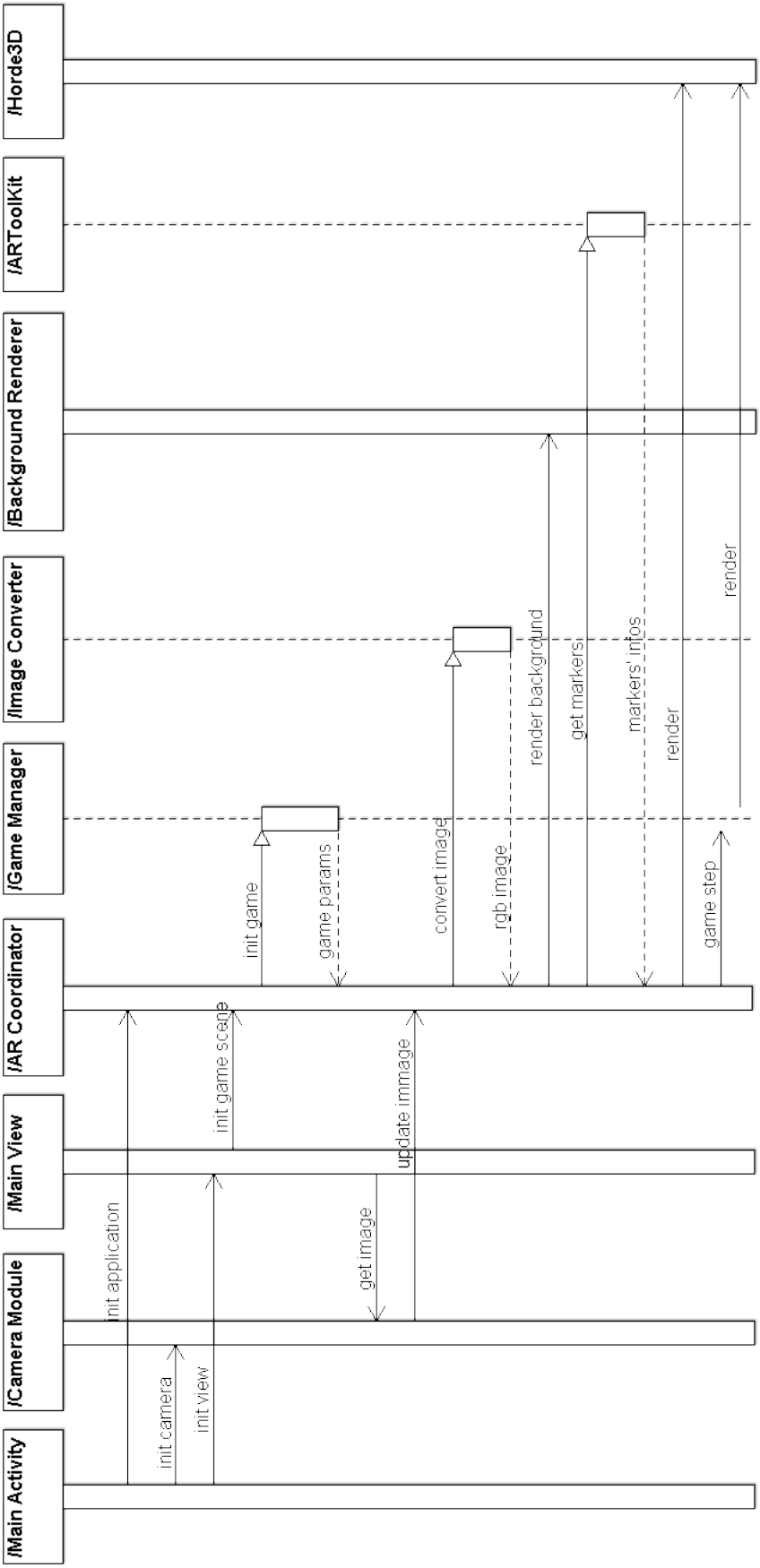

Figura 6.3: progetto: interazioni del sistema

Capitolo 7

Progettazione del game

Ci si occuperà ora di come è stato implementato lo specifico game costruito sopra il sistema di AR che è stato precedentemente definito.

1 Definizione del game

Il game prevede la presenza di una griglia regolare composta da nove basi quadrate, ciascuna delle quale può rappresentare uno fra quattro possibili simboli: una croce ad X azzurra, un cerchio rosso, un quadrato viola ed un triangolo verde (i simboli dei pulsanti di un comune joypad Plastation) oppure di un simbolo vuoto non rappresentante nessuno di questi. Si veda la Fig reffig:texture per averne una rappresentazione chiara. La disposizione dei diversi tipi di simboli nella griglia è definita in maniera del tutto casuale ad inizio partita. Questa griglia è l'elemento che deve venire posizionato esattamente sopra al marker. Contemporaneamente il gioco prevede una graduale e continua comparsa di elementi dalla forma cubica a partire da una altezza prefissata, ciascuno dei quali presenta sulle proprie facce uno dei simboli appena citati, proprio come le stesse basi della griglia. Questi cubi cadono dalla posizione di partenza con un moto accelerato che simula la caduta di un oggetto reale per accelerazione di gravità, e scompaiono una volta raggiunta la griglia. Lo scopo del gioco è di far arrivare piè cubi possibili sopra una casella con lo stesso simbolo raffigurato, nell'ovvia ipotesi che per ogni tipo di cubi esista una base sottostante del tipo equivalente.

Essendo il gioco ottimizzato per il dispositivo XPeria Play, si prevede per questo l'utilizzo dei suoi comandi fisici presenti sul relativo slide-pad. Per altri tipi di dispositivi, che non presentano pulsanti fisici, dovrà ovviamente essere disponibile una serie equivalente di pulsanti virtuali presenti sullo schermo tattile, in grado di svolgere la stessa funzione. Si prevede comunque l'utilizzo di 8 pulsanti in tutto. I 4 pulsanti raffiguranti i quattro simboli e tipici di ogni pad Playstation (quindi anche del dispositivo in esame) servono per decidere quale tipo di cubo è possibile spostare, mentre i pulsanti freccia per muovere effettivamente i cubi selezionati nelle due dimensioni mentre questi cadono sulla griglia. Per evidenziare meglio quando un cubo viene selezionato si è deciso inoltre di utilizzare degli elementi che simulano l'aspetto di quattro fasci di luce, che si dispongono esattamente sulla base sottostante in quel momento.
La scelta del marker da utilizzare è ininfluente ai fini del progetto, quindi uno qualunque va bene. Per il presente caso si è scelto quello già utilizzato nei diversi esempi di ARToolKit, riportante la scritta Hiro come template, presentato anche in Fig. reffig:marker1.

Figura 7.1: marker utilizzato per il gioco

Per capire qual'è l'idea del gioco finalmente implementato e funzionante su device, si faccia riferimento alla Fig. reffig: artower example 2 nel capitolo relativo al deployment.

2 Modelli e risorse utilizzate

In tutto si hanno dieci diversi tipi di modelli per il gioco, dei quali cinque per le basi della griglia (quattro con simboli ed una vuota), quattro per i blocchi a forma di cubo ed uno per i fasci che servono per indicare la posizione di un

blocco sopra ad una base. Ciascun modello renderizzato mediante Horde3D deve essere descritto dagli opportuni file risorsa propri dell'engine, cioè un file per la geometria, un file di materiale ed eventualmente un file relativo al sotto-grafo della struttura ad albero di scena (è possibile anche definirlo tramite programma, ma un approccio di tipo dichiarativo è generalmente preferibile), piè un file per le animazioni ed uno o piè file immagine per le texture, se presenti. Si è deciso di utilizzare un unico file di immagine per le texture contenente tutti i quattro diversi simboli (illustrato in 7.2), ed un diverso file di geometria per ciascuna base e ciascun blocco, ciascuno dei quali con diverse coordinate texture per far riferimento a diverse aree dell'immagine. Un'altra possibile soluzione sarebbe stata quella di utilizzare un solo file per tutte le basi ed uno per tutti i blocchi e quattro diversi file d'immagine, tuttavia, dal momento che non si tratta di modelli geometrici particolarmente complessi, il risultato della scelta è risultato sostanzialmente indifferente.

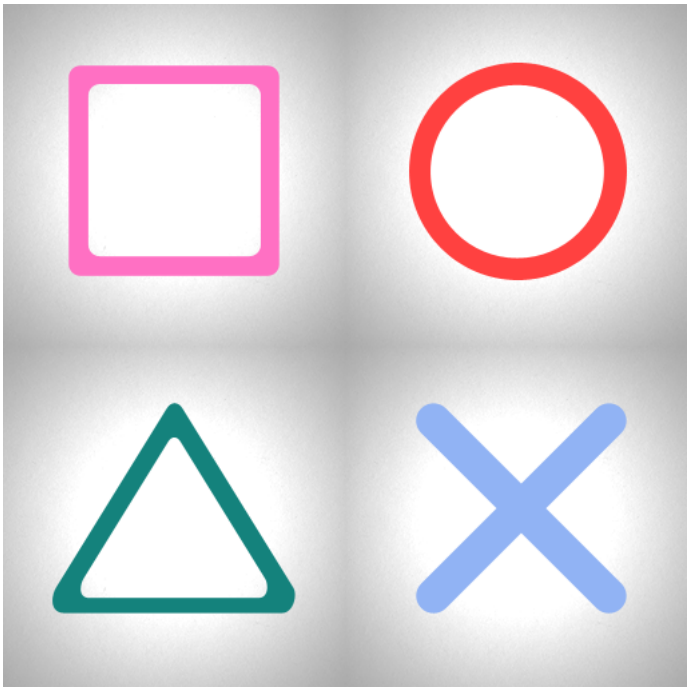

Figura 7.2: texture utilizzata per i blocchi e le basi del gioco

Ricordiamo che i file di geometria, i file di materiale, i file di animazione ed il file del sotto-grafo di scena del relativo modello vengono generati trami-

te l'apposito tool **ColladaConv**, fornito assieme all'intero package di Horde3D, a partire da files di tipo Collada, che contengono tutte l'informazione necessarie. Per la presente tesi è stato utilizzato il software commerciale di modellazione geometrica **Maya** per la creazione dei diversi modelli e quindi dei relativi file Collada.
Viene utilizzato un file risorsa di tipo *scene graph* anche per la definizione dell'elemento griglia, anche se questo non contiene riferimenti a risorse geometriche al suo interno. Infatti, come già indicato, la struttura interna di questa griglia non è definita a priori ma viene inizializzata in maniera casuale. I modelli di tipo base qui contenuti vengono allora aggiunti tramite programma, a tempo d'inizializzazione del gioco, per mezzo delle opportune API del motore grafico. Utilizzando Horde3D la creazione di questo elemento griglia diviene oltretutto abbastanza naturale, poiché è possibile considerarla come un nodo padre nel grafo di scena, ed al quale sono collegati nove nodi figli, ciascuno in una propria posizione prefissata relativa al padre stesso.
Allo stesso modo delle basi che la compongono, anche la posizione dei blocchi che cadono e dei fasci sottostanti deve necessariamente essere relativa alla griglia stessa. In questo modo ci si può preoccupare del posizionamento esatto nel mondo solamente per quest'ultima, ricordando che deve apparire come sovrapposta al marker nell'immagine reale. Questa operazione, come già accennato nel capitolo precedente, può venire effettuata automaticamente dal sistema di AR, una volta registrata la corrispondenza con l'ID del marker che si intende utilizzare. Per il corretto posizionamento occorre però settare anche la matrice di proiezione e questo è possibile solo a livello di shader: il sistema di AR scompone tale matrice in quattro diversi vettori, passati ad altrettante variabili uniform, quindi tutto quello che rimane da fare è ricostruirla. Si riporta di seguito la parte di *vertex shader* che si occupa di questo aspetto.

```
uniform vec4 projection_matrix_1;
uniform vec4 projection_matrix_2;
uniform vec4 projection_matrix_3;
uniform vec4 projection_matrix_4;
uniform mat4 worldMat;
```

```
uniform vec3 vertPos;

...

attribute vec3 vertPos;
attribute vec2 textCoords0;
attribute vec3 normal;

...

varying vec2 vTextCoords
varying vec3 vPos;
varying vec3 vNorm;

...

void main( void )
{
      vPos = vertPos;
      vTextCoords = textCoords0;
      vNorm = normal;

       mat4 projection_matrix = mat4( uniform vec4
          projection_matrix_1, uniform vec4
          projection_matrix_2, uniform vec4
          projection_matrix_3, uniform vec4
          projection_matrix_4 );

       ...

       gl_Position = projection_matrix * worldMat *
          vec4( vertPos, 1.0 );
}
```

Questo vertex shader è lo stesso per tutti i contesti di shader definiti dal-

l'applicazione, e questo a differenza di quanto accade per i fragment shaders come verrà chiarito a breve. Del suo funzionamento nel contesto dell'intera pipeline di rendering ne forniamo anche uno schema in 7.3. Da notare come avvenga direttamente il passaggio dalla traformazione nel sistema di coordinate mondo alla trasformazione prospettica, questo perchè, essendo il punto di vista definito nell'origine e rivolto in direzione entrante, la trsformata nel sistema di coordinate di vista non viene considerata.

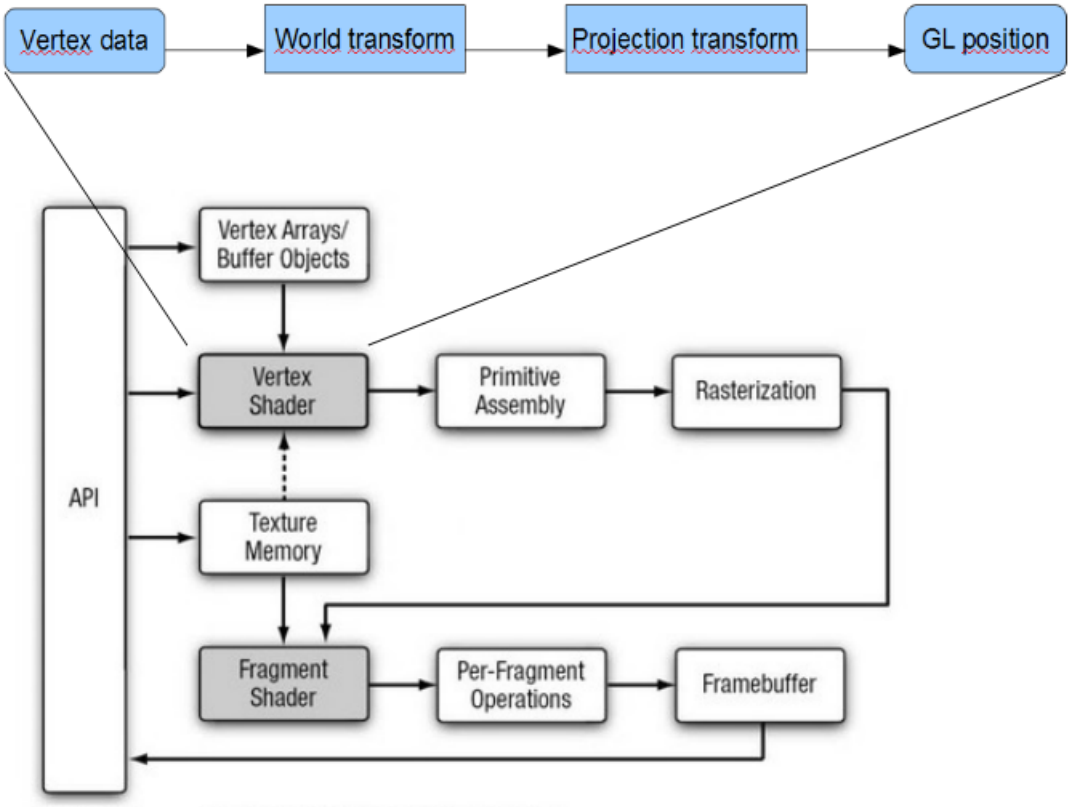

Figura 7.3: schema delle operazioni del vertex shader utilizzato

3 Illuminazione e rendering finale

Rimane da definire in che modo la scena finale viene renderizzata, in particolare con quale modalità di shading. Per un maggiore realismo si è deciso di adottare il modello d'illuminazione con *Phong Shading*, quindi determinando

i colori a livello di fragment shaders ed utilizzando i valori delle normali interpolate da vertice a vertice per calcolare il livello d'influenza di ogni raggio luminoso in un dato punto. L'alternativa sarebbe stata di utilizzare il *Gouraud Shading*, ma i risultati ottenuti sarebbero stati certamente meno gradevoli e comunque la scena da renderizzare non è particolarmente complessa per giustificarne l'uso al posto dell'altra modalità. In ogni caso qualunque tipo d'illuminazione è stata basata sul modello di Phong, descritto dalla seguente formula:

$$I = Ke + Ia * Ka + Id * Kd + Is * Ks \tag{7.1}$$

dove **Ia**, **Id** ed **Is** rappresentano rispettivamente le somme dei contributi ambientali, diffusivi e speculari di tutte le sorgenti luminose, mentre **Ke**, **Ka**, **Kd** e **Ks** i coefficienti emissivo, ambientale, diffusivo e riflessivo di ciascun materiale, che devono avere valori compresi fra 0 ed 1 (non inclusi). Almeno in questo caso comunque il contributo emissivo del materiale può venir ignorato. Horde3D fornisce un determinato supporto per effettuare il rendering con d'illuminazione virtuale della scena. La strategia utilizzata prevede l'associazione di ogni sorgente luminosa, rappresentata da un nodo nello scene graph, ad un dato contesto di shader il quale determina la sua influenza sul colore finale di un qualunque punto. Per simulare dunque un effetto illuminazione viene effettuato un passo di rendering per ogni sorgente luminosa presente, utilizzando appunto questi contesti di shader. L'engine associa poi automaticamente i valori di colore, raggio d'influenza, intensità ed altri parametri a determinate variabili uniformorm richiamabile all'interno degli shader stessi. Esistono due diversi comandi, richiamabili all'interno di una risorsa di tipo pipeline, che permettono di effettuare il rendering della scena in questa maniera, e si differenziano leggermente per le modalità esatte con cui il relativo processo avviene. Nel nostro caso verrà utilizzata un'illuminazione di tipo *forward lighting*, perchè più adatta al caso con poche sorgenti di luce in scena.

Non rimane allora altro da fare che definire un'opportuna sorgente luminoisa (o più di una se necessario) in scena e associarla ad un nodo sullo scene graph e ad un determinato contesto di shader. Come già detto, il contributo d'illuminazione andrà calcolato a livello di fragment shader, sulla base dei valori

delle normali interpolate fra vertici e vertici. In questo caso ci vengono fortunatamente in aiuto anche i file di libreria, forniti assieme allo stesso engine e liberamente utilizzabili, fra i quali anche uno che fornisce delle funzione per il calcolo del contributo d'illuminazione basato sul modello di Phong. Riportiamo di seguito la funzione specifica che è stata utilizzata.

```
uniform vec3 viewerPos;
uniform vec4 lightPos;
uniform vec4 lightDir;
uniform vec3 lightColor;

vec3 calcPhongPointLight( const vec3 pos, const vec3
   normal, const vec3 albedo, const float specMask,
   const float specExp )
{
        vec3 light = lightPos.xyz - pos;
        float lightLen = length( light );
        light /= lightLen;

        // Distance attenuation
        float lightDepth = lightLen / lightPos.w;
        float atten = max( 1.0 - lightDepth *
           lightDepth, 0.0 );

        // Lambert diffuse
        float NdotL = max( dot( normal, light ), 0.0 );
        atten *= NdotL;

        // Blinn-Phong specular with energy
           conservation
        vec3 view = normalize( viewerPos - pos );
        vec3 halfVec = normalize( light + view );
        float spec = pow( max( dot( halfVec, normal ),
           0.0 ), specExp );
```

```
        spec *= (specExp * 0.04 + 0.32) * specMask;   //
            Normalization factor (n+8)/8pi

        // Final color
        return albedo * lightColor * atten * shadowTerm
            * (1.0 + spec);
}
```

La funzione **calcPhongPointLight** viene invocata dal *fragment shader* per conoscere il colore finale di un fragment, decidendo anche quali valori passare in input. Le variabili di tipo uniform elencate in alto sono invece fra quelle dichiarate e modificate automaticamente dall'engine grafico sulla base delle impostazioni delle luci e della posizione della camera.
E' da notare come nel calcolo si tiene conto dei diversi tipi di contributi, ad eccezione di quello ambiente. Questo perché si è deciso di dividere il processo di rendering relativo all'illuminazione virtuale in due passaggi distinti, effettuando prima il calcolo del contributo ambientale delle luci in scena, approsimato con un semplice calcolo del colore senza illuminazione virtuale ed utilizzando un contesto di shader differente, poi il processo di *forward lighting* vero e proprio. Questa scelta è motivata anche dal fatto che, in Horde3D, il processo di rendering tramite sorgente luminosa esclude automaticamente qualunque geometria che si trovi fuori dal raggio d'azione della luce stessa (e questo per motivi di ottimizzazione), quindi è spesso preferibile effettuare anche un passaggio di rendering a parte che coinvolga tutti modelli in scena. Da notare anche di come si tiene conto dell'effetto di riduzione della luminosità con l'aumentare della distanza dal centro della sorgente.
Riportiamo in seguito anche la parte di fragment shader dove questa funzione viene richiamate.

```
uniform sampler2D texture;
uniform vec4 color;
uniform float specMask;
uniform float specExp;

...
```

```
varying vec2 vTextCoords
varying vec3 vPos;
varying vec3 vNorm;

...

void main( void )
{
        vec3 albedo;
#ifdef _F01_texture
        albedo = texture2D( texture, vTextCoords );
#else
        albedo = color;
#endif

        vec3 pos = vPos;
        vec3 norm = vNorm;

        ...

        gl_Color = calcPhongPointLight( pos, norm,
           albedo, specMask, specExp );
}
```

Le variabili uniform qui riportate vengono caricate direttamente dall'engine, una volta che vengono dichiarate ed instanziate all'interno di un file di tipo material (per la variabile **texture** bisogna far riferimento alla risorsa di tipo texture), mentre per le variabili varying si faccia riferimento al precedente frammento di vertex shader. **F01 texture** è invece un flag, utilizzabile in un contesto di *uber-shaders*, qui definito e usato per l'abilitazione delle texture. In Fig. **??** viene evidenziato il funzionamento del fragment shader nel contesto della pipeline.
Si tenga a mente che il fragment shader riportato è quello relativo al contesto di shader associato alle sorgenti luminose in scena e quindi utilizzato per

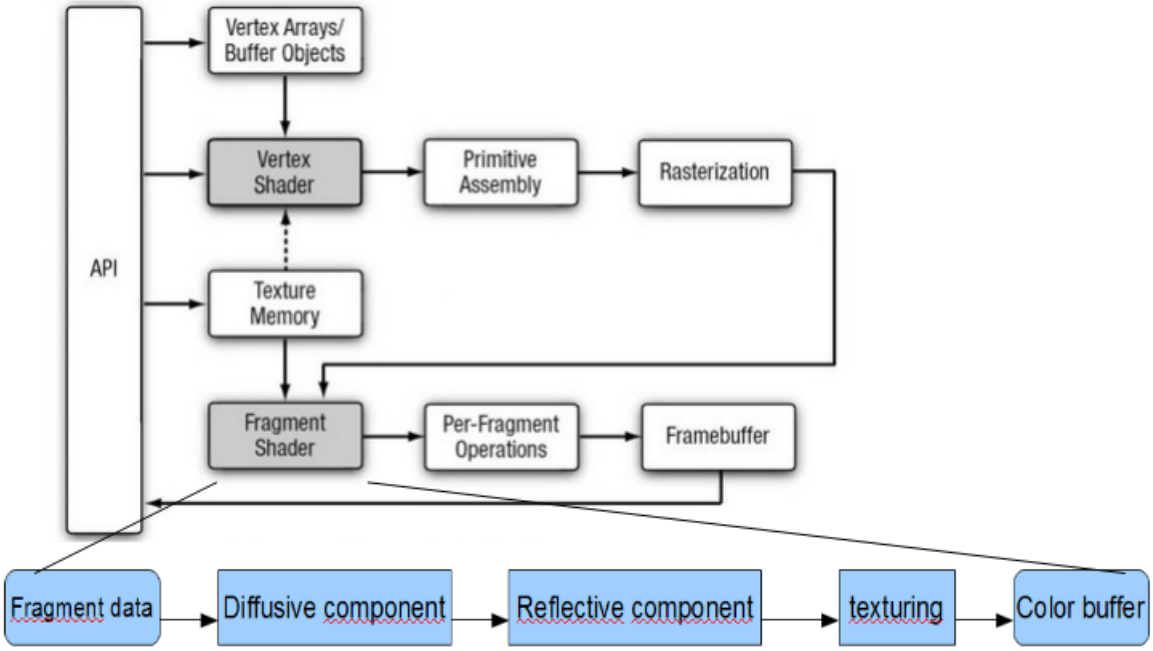

Figura 7.4: fragment shader per il rendering con illuminazione virtuale

effettuare il rendering con Phong shading. E' stato detto che il contributo d'illuminazione ambientale viene eseguito invece in un passo precedente ed indipendente, quindi un diverso fragment shader viene usato, che si limita ad implementare un metodo di colorazione senza illuminazione virtuale.

```
uniform sampler2D texture;
uniform vec4 color;
uniform float ambientIntensity;

...

varying vec2 vTextCoords

...

void main( void )
```

```
{
        vec3 albedo;
#ifdef _F01_texture
        albedo = texture2D( texture, vTextCoords );
#else
        albedo = color;
#endif

        vec3 pos = vPos;
        vec3 norm = vNorm;

        ...

        gl_Color = ambientIntensity * color;
}
```

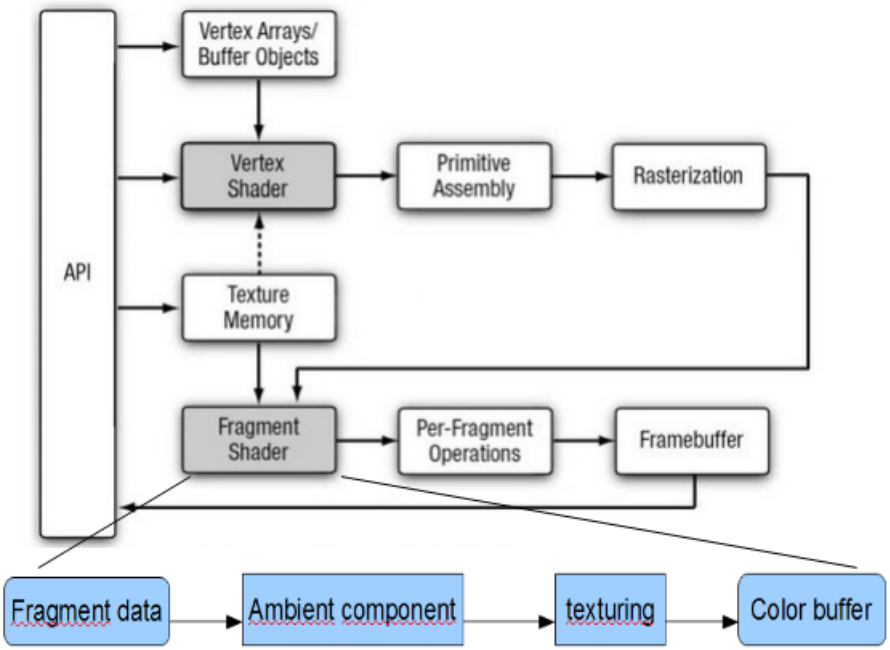

Figura 7.5: fragment shader per il rendering senza illuminazione virtuale

Esiste anche un ulteriore passaggio di rendering, che serve per renderizza-

re gli unici oggetti trasparenti presenti in scena, ossia i fasci che servono per delimitare la base sottostante ad un blocco in caduta. Anche per questo passaggio viene utilizzato lo stesso fragment shader appena mostrato, con l'eccezione che questa volta, mentre viene definito il relativo contesto di shaders, viene richiesto all'engine di utilizzare una diversa modalità di blending per il colore dei singoli pixels, in mdo da tenere conto del colore già presente e della componente alpha del nuovo colore. Ovviamente, affinché ciò funzioni, è anche necessario che questa modalità di rendering venga effettuata solo in seguito alle altre due.

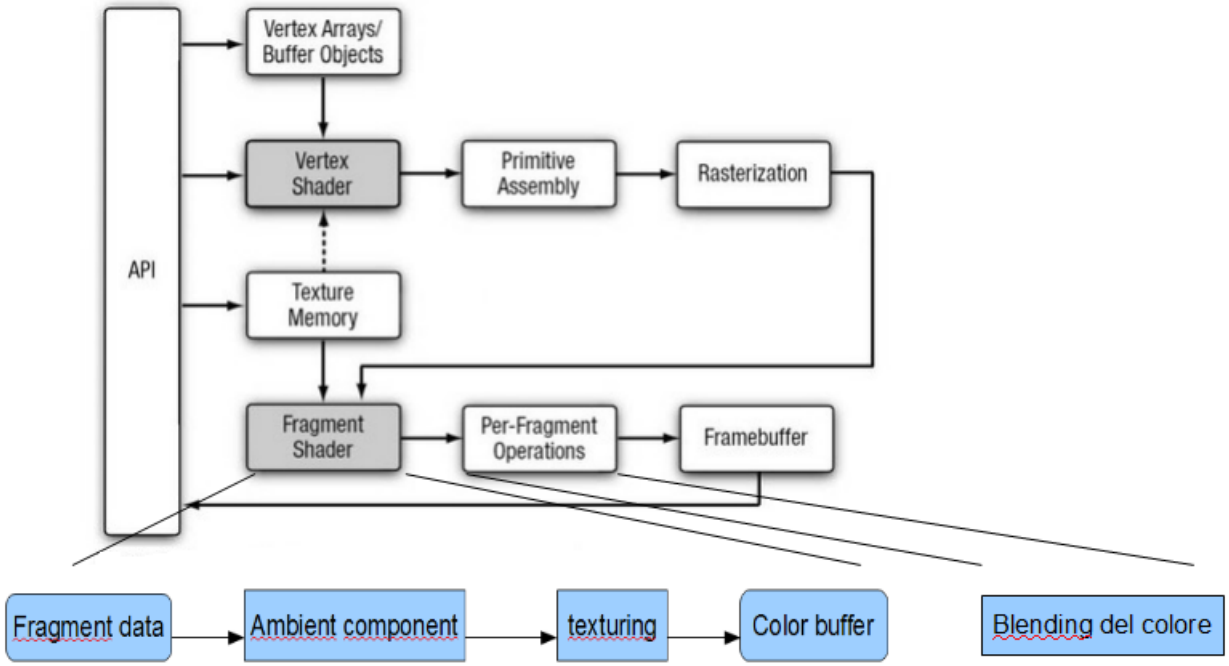

Figura 7.6: fragment shader per il rendering (senza illuminazione virtuale) di superfici trasparenti

L'intera fase di rendering si svolge dunque in tre diversi passaggi, definiti ciascuno a livello di pipeline, rispettivamente per la resa del contributo d'illuminazione ambientale, per il rendering delle rimanenti componenti d'illuminazione e per il rendering delle superfici trasparenti. Se si considera anche il procedimento di rendering del background, che non viene effettuato a livello applicativo ma a livello di sistema e senza l'utilizzo del motore grafico allora

lo schema complessivo di funzionamento si traduce in quello in Fig. 7.7. In 7.8 abbiamo invece uno schema complessivo dei contesti di shaders utilizzati per la pipeline. Va ricordato che la procedura di rendering tramite illuminazione con Phong shading viene effettuata una volta per ogni luce presente in scena. Nell'immagine sono stati evidenziate anche le procedure di pulizia del color buffer e del depth buffer, ed è da notare che la prima viene già invocata dal sistema di AR all'inizio di ogni ciclo, mentre l'altra viene invocata in un secondo momento per permettere la scrittura sui pixel sui quali è già stato renderizzato il background della scena.

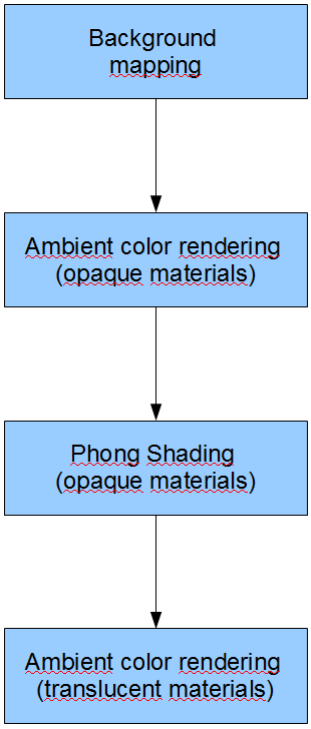

Figura 7.7: schema a pipeline dei vari procedimenti di rendering

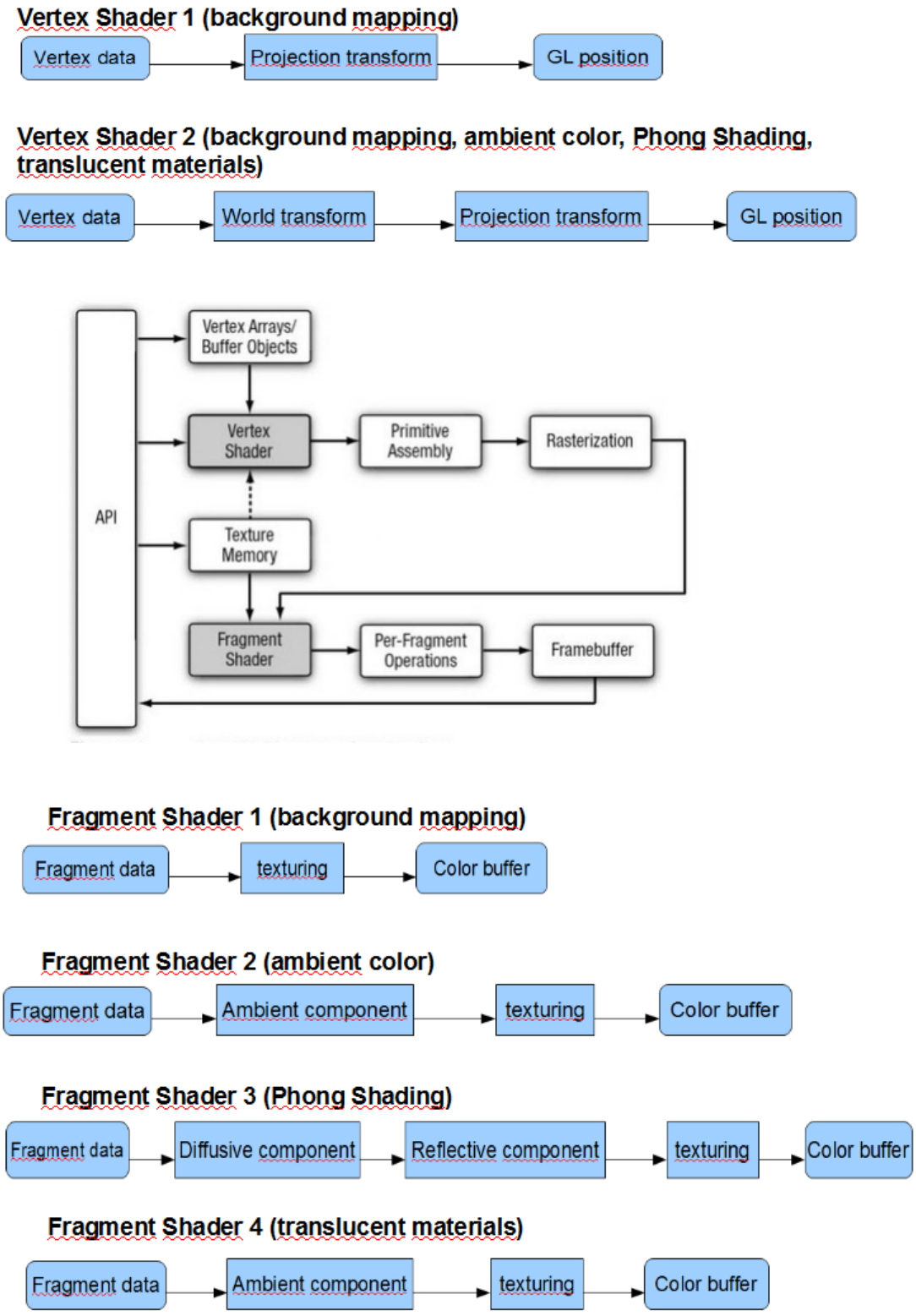

Figura 7.8: schema dei contesti di shaders utilizzati per la pipeline di rendering

Capitolo 8

Deployment e test

L'obbiettivo finale del progetto è ovviamente quello di verificare il funzionamento dell'applicazione su dispositivo reale, ragion per cui la fase *deployment* e di *testing* viene normalmente considerata successiva a quella di progettazione. In realtà non vi è una netta distinzione fa le due parti le quali potrebbero andare piè volte a sovrapporsi. Questo perché c'è spesso la necessità di testare le singole funzionalità di un software, man mano che queste vengono implementate, che contribuiscono al funzionamento complessivo. Anche nel nostro caso è stato seguito un ben determinato programma di testing, mentre si procedeva con la progettazione dell'intero sistema, partendo dalla verifica dell'immagine di sfondo, al test delle librerie e della loro corretta integrazione, alla verifica dei vari effetti di rendering desiderati, fino ad ottenere il risultato finale desiderato.

1 Deployment su dispositivo

Come già discusso in precedenza, per portare un'applicazione Android su dispositivo reale o virtuale è prima necessario compilare i file sorgenti, linkarli e quindi generare il file *.apk*. Inoltre, nel caso di utilizzo di parti in codice nativo, è prima necessario generare i file di libreria tramite i tool dell'NDK e likare anche quest'ultimi assieme agli altri. Per agevolarci da questo complesso processo, per il corrente progetto abbiamo fatto uso del plugin **ADT** (*Android Development Tool*) del framework **Eclipse**, che permette di eseguire in maniera automatica tutte queste procedure ed agevola nella gestione

delle varie funzionalità per il deployment ed il testing. Ad esempio permette la visualizzazione, in un'apposita finestra, di tutti i messaggi del **logcat**, che è il principale strumento di debug in Android. Terminata la procedura di generazione (e di firma crittografica), il file .apk della nostra applicazione è stato importato nel device fisico in esame tramite gli opportuni comandi d'installazione. Tramite i tool di Android sarebbe stato anche possibile l'installazione in un device virtuale, tuttavia nel nostro caso questo non ne avrebbe permesso il funzionamento poiché funzionalità come la telecamera (ovviamente) o l'utilizzo delle OpenGL ES di versione 2.0 non sono disponibili su emulatore. Inoltre si ha la necessità, esplicitamente dichiarata in fase di requisiti, di ottimizzare il funzionamento del game su un determinato modello di smartphone.

2 Test d'esecuzione

Come già detto la fase di testing è stata eseguita in piè tempi, testando di volta in volta le singole funzionalità

Dapprima si è valutato il modo in cui veniva renderizzata l'immagine reale apposta su texture di sfondo. E' stato necessario tarare piè volte le coordinate texture per ottenere i risultati desiderati fare in modo che l'immagine si adattasse perfettamente allo schermo. Inoltre è stato necessario assicurarsi che la funzione di conversione del formato fosse implementata correttamente, affinché la texture risultasse uguale all'immagine reale. Il vero problema si è però rivelato essere una questione di performance, dal momento che, pur essendo l'immagine visualizzata in maniera corretta, risultava evidente un notevole calo nel frame-rate di rendering. Ciò era dovuto alla pesantezza dell'immagine passata come contenuto della texture, e conseguentemente alla grandezza della texture stessa, dal momento che per default Android utilizza la massima risoluzione disponibile per la camera, che nel caso del dispositivo utilizzato è di **1200x800**. Una dimensione così elevata per l'immagine andava inoltre ad aumentare inesorabilmente il tempo richiesto dalla funzione di conversione in formato RGB, per quanto questa fosse ottimizzata. Sono stati effettuati utilizzando risoluzioni meno elevate, riscontrando effettivamente decisi miglioramenti prestazionali, a fronte purtroppo di una qualità visiva

piè bassa. Come verrà anche evidenziato in seguito, sono stati necessari piè tentativi per cercare il giusto compromesso.

Per verificare il corretto funzionamento ed integrazione della libreria ARToolKit si è semplicemente fatto uso delle funzionalità offerte dal **logcat** di Android, stampando in debug tutte le informazioni relative a ciascun marker individuato. Apparentemente nessun problema è apparso in questa fase. Quindi si è testata anche l'integrazione con la componente grafica, verificando che le funzionalità del motore grafico risultassero effettivamente utilizzabili e che i modelli venissero posizionati esattamente sopra i marker. Prima di utilizzare i modelli di gioco veri e propri, sono stati utilizzati modelli di prova, non integrati nel contesto di un game ma con il solo scopo di venire posizionati nella posizione desiderata. Nella figura 8.1 viene ad esempio mostrato il modello di un personaggio umano, di colorazione uniforme ma dotato anche di un animazione (gestita secondo le modalità dell'engine Horde3D) posizionato esattamente sopra il marker.

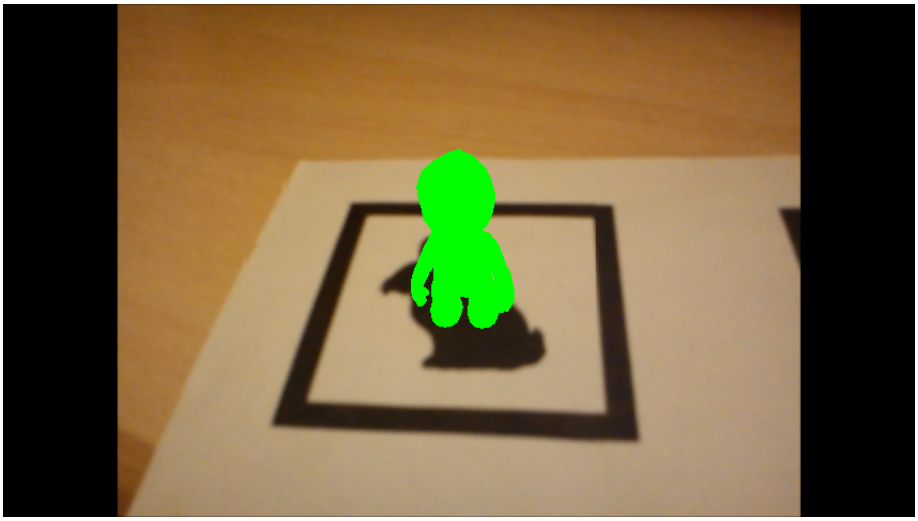

Figura 8.1: visualizzazione di un modello d'esempio per l'integrazione del motore grafico nel sistema

In figura 8.2, illustriamo infine il gioco vero e proprio, già implementato ed in esecuzione.

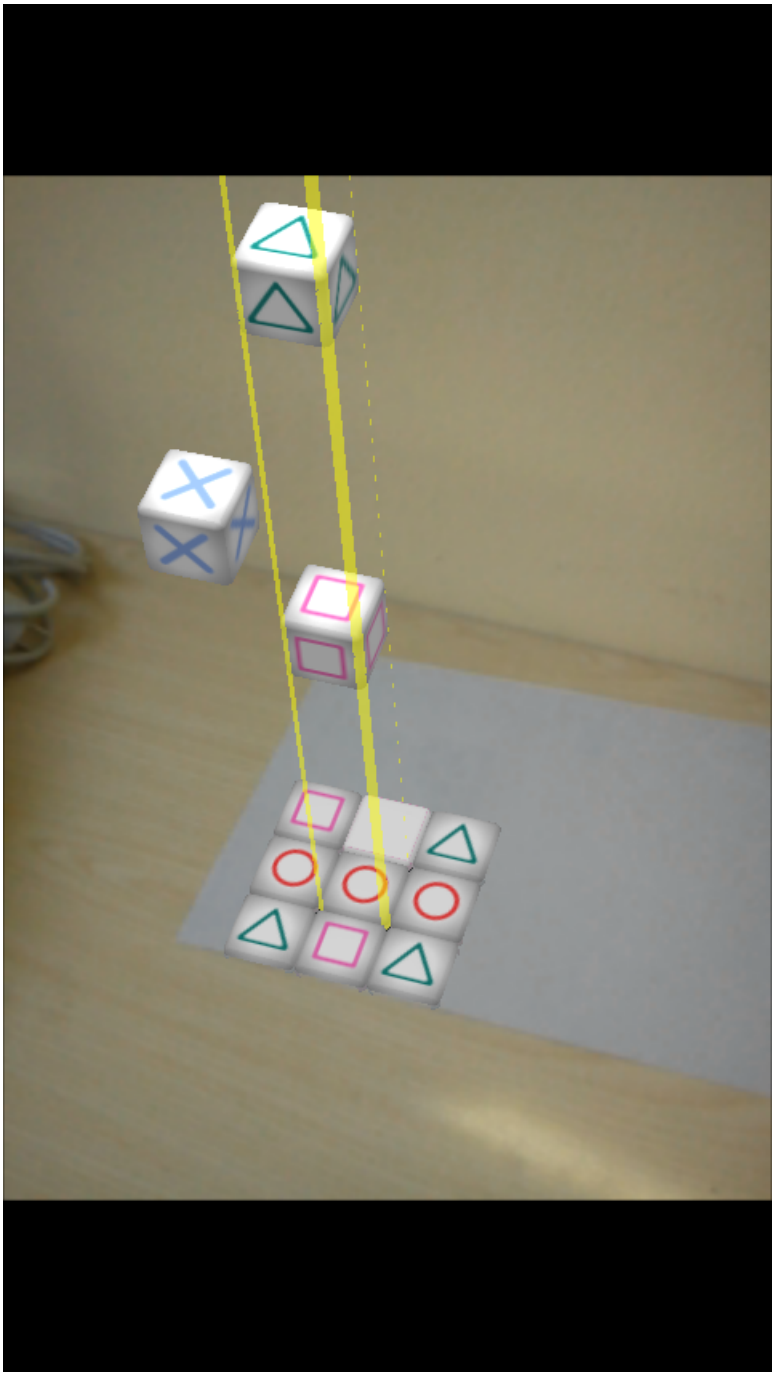

Figura 8.2: il game di progetto ultimato in esecuzione su device

3 Miglioramenti

3.1 Gestione della risoluzione dell'immagine

Come già detto la risoluzione utilizzata per la camera e la conseguente dimensione del buffer immagine incide pesantemente sulle prestazioni, ragion per cui operare una regolazione a riguardo è certamente d'obbligo. Anche scegliere una risoluzione troppo bassa però (si potrebbe per semplicità scegliere quella piè bassa fra tutte quelle disponibili) non si rivelerebbe una scelta appropriata, non solo per una questione di qualità visiva ma anche perché da ciò dipende la precisione del processo di riconoscimento dei markers. Occorre giungere ad un giusto compromesso. Vi è però un duplice problema: primo che ogni diverso tipo di device possiede diverse risoluzioni per la propria camera, ed una scelta a priori potrebbe non risultare fra quelle disponibili, secondo che anche la potenza di calcolo è differente, quindi una risoluzione molto elevata potrebbe essere facilmente gestita da dispositivi piè avanzati e risultare così piè adatta allo scopo. Android permette di conoscere quali sono tutte le risoluzioni disponibili e di sceglierne una fra queste, ma occorre ovviamente stabilire almeno un criterio di scelta.

A questo punto si hanno due possibili strategie: la prima è di fissare un valore minimo e scegliere la risoluzione piè piccola ma che rispetti almeno tale valore, in modo da ottenere almeno una qualità accettabile dell'immagine, mentre per contro la seconda prevede di utilizzare un valore di massimo e di scegliere quella piè grande finché possibile, così da non rischiare un eccessivo degrado delle prestazioni. Nel caso non fosse possibile rispettare tali valori si scelgono rispettivamente la risoluzione minore o maggiore disponibile. Dal momento che si è ritenuto che mantenere un buon frame-rate sia fra i due il requisito prioritario, si è deciso di seguire questa seconda opzione. E' bene allora che il valore scelto garantisca un buon frame rate, in modo che le prestazioni rimangano accettabili anche per dispositivi meno potenti di quello in uso. Non si avranno certamente le stesse performance ma con ogni probabilità si rimarrà comunque entro i limiti di tollerabilità Dopo vari tentativi si è deciso di adottare come risoluzione ideale una di valore **320x240**. Seppur la qualità dell'immagine risulta evidentemente inferiore al caso di utilizzo della risoluzione massima, i risultati rimangono ancora accettabili. Inoltre l'uso di

un valore ancora inferiore non avrebbe senso perché non produrrebbe risultati visibilmente migliori.

3.2 Migliore rilevamento dei markers

L'utilizzo di un risoluzione inferiore a quella ottima incide negativamente sulla funzionalità di individuazione dei markers da parte della libreria ARToolKit, poiché l'immagine catturata appare meno nitida ed i contorni delle figure meno definiti. Occorre allora applicare degli accorgimenti. Anzitutto vanno rivisti alcuni parametri di configurazione dello stesso ARTooKit, in particolare bisogna assicurarsi che l'algoritmo utilizzato sia quello piè efficace possibile, ossia l'RPP (*Round Planar Pose*). E' ovviamente l'algoritmo piè oneroso in senso computazionale, ma in ogni caso il vantaggio ottenuto utilizzando una risoluzione minore compensa ampiamente il dispendio di risorse.
Inoltre vanno settati determinati parametri in modo da impedire al sistema di effettuare un'operazione di smoothing aggiuntiva sull'immagine, prima del processamento vero e proprio. Si tratta di un'operazione di *image processing* che generalmente evita di avere pixel troppo dispersi attorno ai contorni e quindi aiuta a definire meglio le forme, tuttavia già con l'utilizzo di immagini a bassa risoluzione non si rende particolarmente necessaria.
Infine occorre gestire con cura un altro parametro, cioè il valore di soglia di binarizzazione per le immagini in grayscale. Infatti ARToolKit necessita di avere immagini in bianco e nero per il rilevamento delle forme, ed il colore bianco o nero di ogni singolo pixel viene deciso proprio in base alla sua tonalità di grigio, che questo risulti inferiore o superiore a tale soglia. Inizialmene si è scelto un valore fissato, però appare subito chiaro che sarebbe bene effettuare un adattamento dinamico in base alle condizioni d'illuminazione dell'immagine, ossia alla quantità di bianco e nero contenute. ARToolKit può essere impostato affinché, ad ogni operazione di riconoscimento, adatti automaticamente la soglia di binarizzazione in base a tali condizioni, tuttavia si è preferito un approccio differente che permettesse di ottimizzare ulteriormente i tempi. La funzione di conversione dell'immagine è stata allora modificata in modo tale da restituire anche il minimo ed il massimo valore di grigio disponibile, in base ai quali effettuare le giuste regolazioni sul valore di soglia,

cioè¨ effettuandone una media pesata. E' un metodo che in alcune occasioni può portare a risultati fallaci, ad esempio nel caso di effetti di bagliori che fanno registrare anomali picchi di colore bianco, ma rimane generalmente assai efficace. Ulteriori operazioni di processamento o valutazione potrebbero per altro incidere negativamente sulle prestazioni.

3.3 Effetto di blending del colore ambiente

Un espediente utilizzabile, nel contesto della realtà aumentata, con lo scopo di massimizzare il senso di realismo, ovvero l'interazione dell'oggetto virtuale nell'immagine reale, è quello di regolare l'illuminazione dell'oggetto virtuale stesso in base all'illuminazione reale percepita da telecamera. Esistono diverse tecniche a riguardo, a seconda dell'effetto desiderato. E' possibile ad esempio individuare una sorgente di luce inquadrata nell'immagine, valutarne approssimativamente intensità ed altri parametri e quindi aggiungerne una virtuale analoga, rilevare le zone in ombra su un qualunque oggetto reale e calcolare la direzione di un'eventuale luce direzionale da aggiungere in scena, e così via. Alcuni esempi di utilizzo di queste tecniche sono approfonditi in [15], [16], [17], [18] e [**?**].

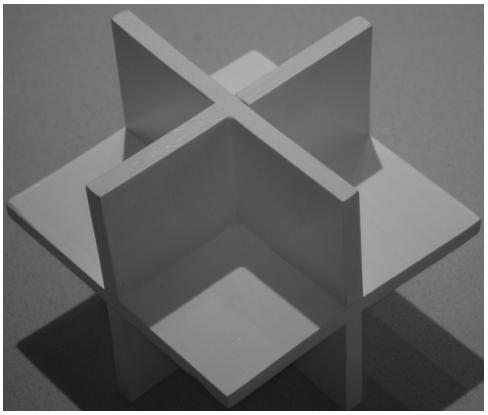

Figura 8.3: esempio di calcolo, da immagine reale, delle sorgenti di luce. In questo caso un oggetto fisico presente nell'immagine esterno viene usato per l'individuazione delle ombre.

In questo contesto ci siamo limitati ad implementare una tecnica assai piè semplice, che però contribuisce comunque ad aumentare il senso di realismo

complessivo. La tecnica si basa su una caratteristica fisica degli oggetti per come il nostro occhio percepisce il loro colore: quando oggetti di colori diversi (e nettamente contrastanti fra loro) appaiono vicini, ognuno tende a diffondere il suo colore sugli oggetti circostanti. Se ad esempio un oggetto di colore rosso si trova a contatto con uno di colore bianco, l'oggetto di colore bianco tenderà ad assumere un'aura di colore rosso. Si dice in questo caso che si ha un **blending** di un colore su una superficie. Si tratta in vero di effetto appena percettibile ma comunque reale.
L'implementazione adottata è per simulare questo fenomeno piuttosto approssimativa, anche se efficace: l'effetto di luce ambientale è simulato tramite una serie di luci di tipo puntiformi, dall'effetto molto soffuso e raggio d'azione piuttosto corto, ma poste assai vicino a ciascun oggetto virtuale da illuminare, e ciascuna di queste offre un diverso contributo di colore a seconda degli oggetti circostanti. In tutto vengono usate esattamente otto luci per oggetto, perché si considerano le otto possibili direzioni dalle quali proviene la luce ambientale. Per prima cosa si individua la porzione quadrata di area dell'immagine entro la quale è contenuto il marker e le otto restanti aree dell'immagine, delimitate dai vertici dell'area stessa, poi si considera il valore medio di colore presente in ciascuna di esse, e questo regola il colore della relativa sorgente di luce. Ricavare i limiti dell'area centrale è assai semplice poiché ARToolKit fornisce, fra le altre cose, anche le coordinate nello spazio bidimensionale dei vertici di ciascun marker individuato, quindi basta considerare i valori estremi di questi. Un aspetto a cui bisogna però prestare attenzione è che, anche se le luci devono rimanere ad una certa distanza dall'oggetto e tutte ad uguale altezza (quindi su un unico piano), queste non devono risultare ruotate nello spazio di vista, poiché ciascuna di esse è univocamente assegnata ad una porzione d'immagine attorno al marker.
In figura 8.4 è mostrato il risultato di questo nuovo effetto, con il blending di un oggetto rosso reale sulla superficie bianca di quello virtuale. Per rendere meglio evidente l'effetto è stato utilizzato un modello virtuale particolare, dalla forma regolare, ossia un dodecaedro, le cui superfici piane sono meglio osservabili di altri oggetti.

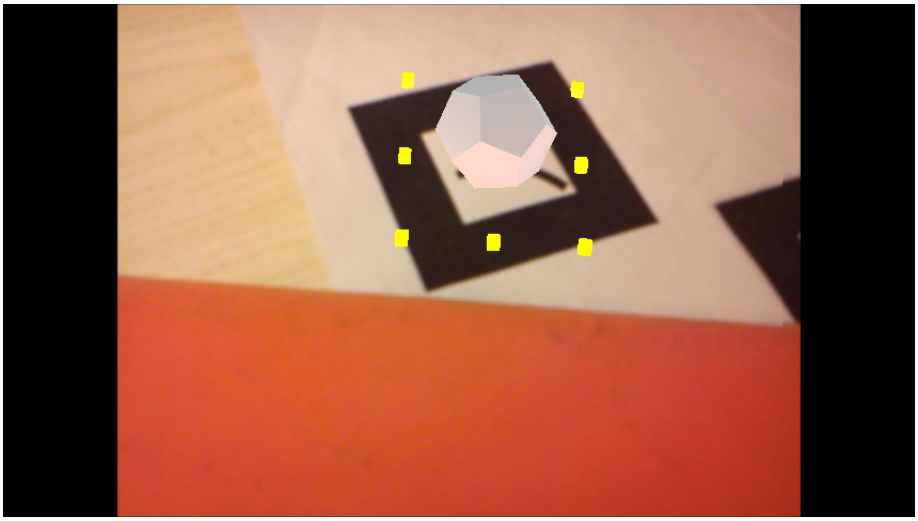

Figura 8.4: effetto di blending ambientale, evidenziato su un modello di dodecaedro, dovuta alla presenza di una superficie rossa estesa nell'immagine. Gli otto piccoli modelli gialli puntiformi, disposti attorno al modello del dodecaedro, servono per indicare la posizione della luci che simulano l'effetto di blending ambientale

Capitolo 9

Conclusioni

Con il presente progetto di tesi possiamo dire di essere andati a fondo non di uno solo, ma di più argomenti inerenti al campo dell'ICT. Abbiamo esaminato e compreso cos'è la **realtà aumentata** e com'è possibile implementarla, abbiamo visto quali possono essere le potenzialità tecniche di un moderno smartphone (e di dispositivi analoghi), anche attraverso la conversione di un vero e proprio motore grafico inizialmente pensato per sistemi home, ed in particolare abbiamo testato le potenzialità di un dispositivo dotato di sistema Android e le molte funzionalità che questo sistema operativo mette a disposizione. Sono certamente tutte tematiche in grado di suscitare grande interesse al giorno d'oggi, anche perchè riguardano in particolare sia un mercato tutt'ora in forte espansione come quello mobile, che uno da sempre al centro di forti interessi come quello del gaming. Per di più l'applicazione dell'AR ai game, almeno nel campo del mobile, si sta rivelando qualcosa in grado di suscitare parecchio interesse verso gli eventuali utenti e la strada verso nuove idee appare ancora pienamente aperta.

Per riuscire ad ottenere il risultato desiderato con il gioco finito è stato necessario procedere ad un'attenta analisi delle varie funzionalità richieste ed una approfondita progettazione ed integrazione. Il risultato è stato certamente soddisfacente ed il gioco, seppur non ancora completo nelle sue meccaniche, può essere considerato un buon esempio d'utilizzo della realtà aumentata. Da un certo punto di vista si potrebbe anzi affermare che il progetto è andato addirittura oltre le aspettative iniziali di realizzare qualcosa che fosse una semplice applicazione di tipo game. Infatti, nel tentativo di comporre as-

sieme le diverse parti del sistema, dal riconoscitore dei marker alla gestione della grafica, quel che è emerso è quanto meno l'abbozzo di un vero e proprio framework per la realizzazione di games che fanno uso di realtà aumentata basata su markers, per ambiente Android. Infatti esiste un determinato layer di funzionalità che permette di agevolare notevolmente il lavoro di progettazione, quali ad esempio la gestione trasparente delle matrici di traformazione o l'associazione automatica fra modelli e template di un marker. Ovviamente per essere considerato un vero e proprio framework dovrebbe fornire anche ben altre funzionalità, come ad esempio un insieme ben definito e documentato di API, ma quanto fatto è comunque un buon punto per eventuali miglioramenti futuri.

Inoltre si hanno a disposizione da una parte la stabilità di un sistema per il riconoscimento di markers quale **ARToolKit**, e dall'altra tutte le potenzialità di un vero e proprio motore grafico per videogiochi di ultima generazione quale **Horde3D**. Proprio per quanto riguarda l'engine grafico però va detto che, pur essendo stato scelto per le sue potenzialità e la sua semplicità, si tratta di un software ancora nella sua versione beta, quindi certamente ancora destinato ad essere soggetto a cambiamenti ed in attesa di un rilascio che possa essere ritenuto totalmente stabile. Un futuro riadattamento per il nostro prodotto, che questa volta risulterà sicuramente meno oneroso, sarà allora probabilmente assai gradito.

Si ricordi anche che, nonostante sia Android la piattaforma di sviluppo scelta come riferimento, una particolare cura è stata posta nel cercare di adottare per quanto possibile un approccio di tipo platform-independent, per cui l'applicazione possa eventualmente in futuro essere portata anche su altri sistemi, come ad esempio iOS. Si tratta senz'altro di una strategia assai gradita nell'ottica di sviluppo di videogiochi da parte di una qualunque impresa, perchè permette di minimizzare il costo in risorse pur sviluppando su più sistemi possibili.

In conclusione dunque ci si augura che, anche quando il videogioco sarà completamente ultimato, il lavoro fin qui svolta possa senz'altro tornare ad essere utile per lo sviluppo di altri eventuali games o applicazioni simili inerenti all'uso della realtà aumentata, o d'aiuto per futuri progetti riguardanti lo stesso argomento.

Appendice A

Il dispositivo Sony Ericcson Xperia Play

1 Analisi delle caratteristiche

La conoscenza approfondita dell'architettura fisica di un dispositivo è molto spesso non solo utile ma anche necessaria qualora l'applicazione che si intendesse sviluppare vada a sfruttare in maniera intensiva le risorse a disposizione. Questo almeno quando si intendono raggiungere risultati ottimali. In ambiente Android ciò è generalmente difficile a causa della varietà di dispositivi sul quale il sistema operativo correntemente funziona, ma, almeno per il corrente progetto, abbiamo un ben preciso device da prendere come riferimento ed in base al quale operare per tutte le ottimizzazioni. Come abbiamo detto si tratta del modello **Sony Ericsson Xperia Play**, smartphone prodotto dall'omonima casa produttrice e particolarmente inspirato alle console da gioco Sony **Playstation**, e del quale andremmo ora ad elencare tutte le caratteristiche rilevanti.

- **CPU:** Qualcomm Snapdragon MSM8255 a 1 GHz
- **GPU:** (integrata) Adreno 205, con supporto a OpenGL ES 2.0, OpenGL ES 1.1, OpenVG 1.1 ed EGL 1.3, e fino a 2 moduli texture
- **RAM:** 512 MB
- **memoria interna:** 400 MB
- **supporto di memorizzazione:** suppord a micro sd-card (fino a 32 GB di memoria)

- **sistema operativo:** Android 2.3 Gingerbread
- **input:** touchscreen multi-touch, accellerometro e slider-pad con 13 tasti
- **risoluzione schermo:** display fino a 854x480 MP
- **camera:** camera principale a risoluzione 5.1 MP

Figura A.1: modello del Sony Ericsson Xperia Play

Appendice B

La libreria ARToolKit

1 Introduzione

ARToolKit ï¿½ una libreria software realizzata con lo scopo di agevolare lo sviluppo di applicazioni facenti uso della realtï¿½ aumentata basata sull'utilizzo di markers, tramite una serie di funzionalitï¿½ di *computer tracking*. Sviluppata inizialmente da Hirokazu Kato dell'Istituto della Scienza e della Tecnologia di Nara nel 1999 (si veda la sua prima disgressione in [1]) e successivamente rilasciata dalla HIT lab dell'Universitï¿½ di Washington sotto licenza pubblica ed open-source, ï¿½ divenuta una delle piï¿½ utilizzate a questo scopo, grazie anche al grado di efficacia e efficienza offerti. Dalla data del rilascio ne sono state sviluppate diverse versioni, migliorate ed alcune con funzionalitï¿½ aggiuntive, ed ï¿½ stata resa disponibile per diverse piattaforme quali Windows, Linux, MacOS e Java.
ARToolKit fa uso di markers per realizzare la funzionalitï¿½ di *camera tracking*, ossia per determinare la posizione del punto di vista nella scena, ed utilizza sofisticate tecniche di *pattern recognition* per individuarne, con notevole precisione, la presenza all'interno di un'immagine passata in input. Tale precisione ï¿½ garantita anche grazie alla forma dei markers utilizzati. L'aspetto di un marker ï¿½ infatti quello di un bordo quadrato nero, di spessore arbitrario, con un'altra area quadrata e bianca all'interno. Questa puï¿½ inoltre contenere una figura di qualunque tipo (ovviamente in bianco e nero) detta *template*, che il sistema puï¿½ riconoscere una volta addestrato. In questo modo ï¿½ possibile identificare univocamente un determinato marker

e stabilire un'associazione univoca fra gli stessi markers ed i diversi tipi di modelli.
L'ottimizzazione con cui il sistema ï¿½ stato implementato rendono inoltre l'intera procedura di riconoscimento relativamente molto rapida, tanto da rendere possibile anche il riconoscimento in real-time, requisito praticamente indispensabile per la realizzazione di applicazioni di tipo interattive come ad esempio giochi.
Va sottolineato che ARToolKit, nell'intero contesto di AR, si occupa esclusivamente del riconoscimento dei marker e del tracciamento del punto di vista della camera. Per la gestione dei restanti aspetti riguardanti un'applicazione che utilizza la realtï¿½ aumentata, come ad esempio la grafica, occorre affidarsi anche ad altri mezzi.
In [2], [3], [4], [5], [7] e [**?**] abbiamo altri esempi di successivi progetti di Kato ed sviluppatori affiliati sempre riguardanti ARToolKit. In [9] un progetto per l'utilizzo della libreria in ambito mobile, mentre in [11] e citebib:games4 esempi di veri e propri games sviluppati sulla base del suo utilizzo.

2 Caratteristiche e funzionamento

L'intero processo di riconoscimento parte ovviamente dall'immagine presa da camera e passata in input, che deve necessariamente essere in formato *grayscale* o *RGB* (il quale comunque viene poi convertito in grayscale). Vengono per prime applicate determinate operazioni di pre-processamento, quali ad esempio di *stretching* o di *smoothing*, in base a come sono stati impostati i parametri di configurazione, quindi una operazione di binarizzazione per renderla in bianco e nero, sulla base di un valore di soglia impostato manualmente o calcolato in automatico. L'immagine cosï¿½ ottenuta ï¿½ la base per l'applicazione degli algoritmi di *pattern-recognition* per l'individuazione delle forme trapezoidali qui contenute. Tali algoritmi si basano essenzialmente su meccanismi di *edge detection*, ossia di rilevamento dei lati. In tutto tre diversi algoritmi sono utilizzabili: in ordine di efficacia *normale*, *normale migliorato* e *RPP (Round Planar Pose)*. Quest'ultimo in particolare ha saputo dimostrare risultati veramente buoni, a scapito ovviamente di tempi di calcolo maggiori.

Figura B.1: Diagramma di funzionamento della libreria ARToolKit all'interno dell'intero processo di AR.

Per ciascun marker individuato vengono memorizzate le coordinate nello spazio bidimensionale dei quattro vertici e del centro geometrico, poi, a partire da questi, calcolate le equazioni dei lati e ricavata la matrice di trasformazione dalle spazio delle coordinate oggetto allo spazio delle coordinate vista. Proprio questa matrice ï¿½ cïï¿½ che permette di posizionare gli elementi virtuali in scena in modo tale da farli apparire collocati sopra al marker nell'immagine reale. Per ricavare queste informazioni ï¿½ perï¿½ necessario che il sistema conosca anche le dimensioni che ciascun marker deve avere, sia nel mondo reale che nello spazio OpenGL. A tale scopo vengono utilizzati particolari file di configurazione, uno per ciascun marker, contenenti tutte le informazioni necessarie e caricati tramite gli opportuni metodi. Ogni volta che uno di questi viene invocato, un nuovo marker viene registrato, con un ID univoco associato ad esso.

Se dopo queste fasi dei markers sono stati individuati si procede all'analisi del template interno a ciascun marker, per verificare se questo appartiene ad

uno di quelli registrati. Dunque i file di configurazione dei markers contengono anche l'informazione relativa alla descrizione dei templates, informazione che ï¿½ codificata mediante una matrice di valori interi ciascuno dei quali esprime il rapporto fra quantitï¿½ di bianco e di nero in una determinata area d'immagine del template stesso. Nei file viene inoltre indicato il grado di suddivisione del template in sotto-aree e quindi le dimensioni della matrice stessa. Se un template viene riconosciuto allora l'ID del relativo marker viene ritornato fra le informazioni relative al marker.

I marker contenenti template devono dunque essere prima registrati per poter essere riconosciuti. Tuttavia esistono anche di un altro tipo che vengono riconosciuti automaticamente dal sistema, detti *markers binari*. Il nome deriva dal fatto che, anzichï¿½ una figura qualunque all'interno dell'area interna, questi contengono una griglia quadrata suddivisa in 16 piccoli riquadri bianchi o neri e la cui configurazione determina appunto l'ID di un marker. Ne esistono due tipi, normali e BCH, diversi per il grado di precisione con cui vengono rilevati. In fase d'inizializzazione occorre settare i parametri di ARToolKit per indicare al sistema quali tipi di markers utilizzare.

Oltre ad effettuare il tracciamento del punto di vista relativamente ad un singolo marker, ï¿½ anche possibile prendere in considerazione un insieme di markers alla volta nel calcolo della matrice di trasformazione. Questo torna utile ad esempio a non perdere il tracciamento di un singolo marker nel caso non fosse piï¿½ interamente visibile nell'immagine, poichï¿½ basta conoscere le posizioni degli altri. Ovviamente i file di configurazione sono differenti, ed ï¿½ inoltre richiesto che l'insieme dei markers sia predeterminato e la loro posizione reciproca non cambi.

Oltre ai file per la registrazione dei marker viene utilizzato anche un altro file di configurazione, questa volta relativo alla *calibrazione della camera*. Questo file contiene le informazioni su come effettuare l'operazione di preprocessamento di stretching dell'immagine in modo tale da annullare l'effetto di distorsione della stessa camera. Il suo contenuto dipende dunque dal particolare dispositivo fisico. Con il package di ARToolKit viene generalmente fornito anche un particolare tool che ne permette la generazione tramite una semplice procedura guidata ed interattiva. Un esempio di come questa funzioni ï¿½ illustrato in B.3.

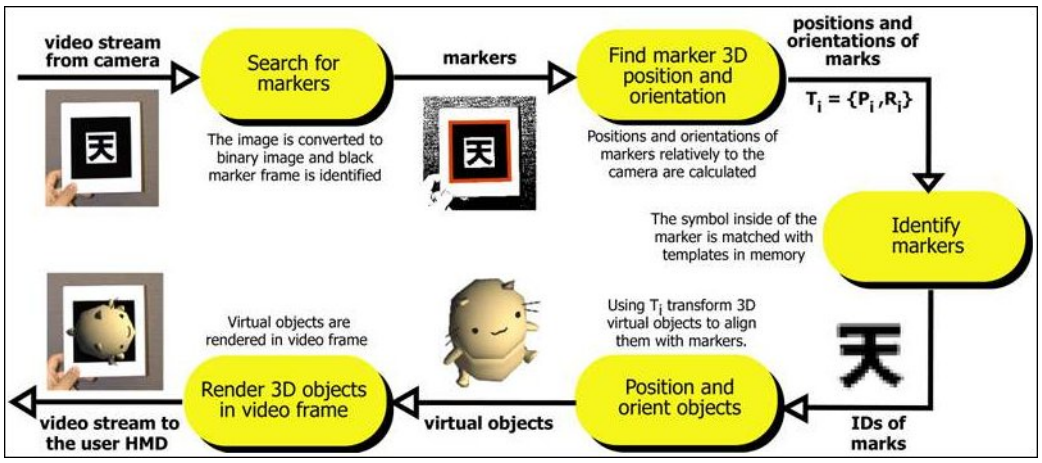

Figura B.2: Diagramma di funzionamento della libreria ARToolKit all'interno dell'intero processo di AR.

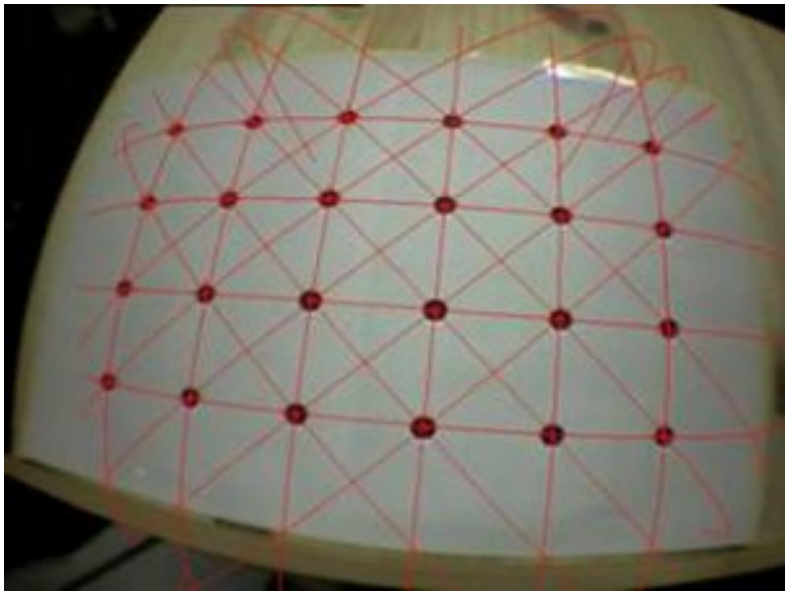

Figura B.3: esempio di utilizzo del tool per la calibrazione della camera: il foglio di carta ed il tavolo di legno appartengono all'immagine reale ma risultano comunque ricurvi a causa dell'effetto di distorsione. Le linee rosse sono invece modellate virtualmente e sono state generate dall'utente tramite successivi click del mause. Tracciate nel giusto modo, con ogni linea che interseca esattamente i suoi punti, il file di calibrazione viene generato correttamente.

Come giï¿½ detto ARToolKit dimostra di avere una grande precisione, oltre

che efficienza, quale libreria per il rilevamento di markers, tuttavia ci sono delle evidenti limitazioni che ï¿½ bene sottolineare. Innanzitutto, almeno nel caso di rilevamento di un marker singolo, un marker deve essere perfettamente visibile e completamente inquadrato dalla camera per essere individuato, per cui anche con una piccola porzione fuori dall'immagine esso non risulta piï¿½ visibile. Questa si rivela in effetti una delle maggiori debolezze di ARToolKit. Inoltre un marker non deve essere neanche eccessivamente distante dall'osservatore, o il relativo template difficilmente verrï¿½ riconosciuto, dal momento che la precisione di riconoscimento ï¿½ strettamente legata alla distanza stessa. Infine ARToolKit ï¿½ particolarmente sensibile anche alle condizioni d'illuminazione, per cui in condizione di luce scarsa il rilevamento potrebbe funzionare in modo meno efficace.

3 Note di porting

Funzionalitï¿½ quali l'accesso diretto al file system non sono stabilmente utilizzabili in ambiente Android, quindi il caricamento diretto del contenuto dei file per la registrazione dei marker e per la calibrazione della camera non dovrebbe essere utilizzabile. Alcuni accorgimenti sono allora stati necessari. Anzichï¿½ effettuare una modifica del codice esistente si ï¿½ perï¿½ preferito attuarne un'estensione, creando nuove classi a partire da quelle esistenti. Nello specifico sono state create le classi **CustomCamera**, a partire dalla classe **Camera**, e **PublicTrackerSingleMarker**, a partire da **TrackerSingleMarker**, le quali possono essere inizializzate anche attraverso il contenuto dei relativi file, caricati usando i metodi opportuni, anzichï¿½ dal suo nome. Per chiarezza un oggetto di tipo **Tracker** ï¿½ quello che si occupa del riconoscimento di un determinato tipo di marker. Il PublicTrackerSingleMarker inoltre permette di conoscere in lettura determinati parametri di configurazione, non conoscibili usando la classe base, aspetto che si ï¿½ rivelato necessario in fase di progettazione.

Appendice C

Il motore grafico Horde3D

1 Introduzione

Horde3D è un piccolo motore grafico in grado però di offrire prestazioni paragonabili a quelle di altri moderni engine, sviluppato da Team Horde3D e distribuito sotto la Licenza Pubblica di Eclipse a partire dal 2006. Il progetto è interamente open source e l'intera collezione di sorgenti è scaricabile anche attraverso il framework stesso di Eclipse. E' interamente scritto in C++ ed è ovviamente stato pensato principalmente per lo sviluppo di games.

Horde3D mette a disposizione un nutrito numero di funzionalità per la realizzazione di diversi effetti grafici, anche tipici di una grafica di ultima generazione, e questo nonostante si tratti di un software di dimensioni relativamente molto ridotte, almeno in paragone alla maggior parte degli attuali motori grafici. Al contempo Horde3D presenta un'interfaccia semplice ed intuitiva ed il suo utilizzo richiede solamente la comprensioni di alcuni semplici concetti. Leggerezza e facilità d'uso sono aspetti che contribuiscono a rendere il motore particolarmente integrabile ed utilizzabile in diversi contesti. Questo anche perché si tratta di un motore puramente grafico, concepito piuttosto per essere usato in combinazione con altri strumenti, per l'implementazione di tutti i restanti aspetti che riguardano un game, come ad esempio la fisica.

Un'altra particolarità di Horde3D è inoltre che si tratta di un engine particolarmente adatto alla gestione di folle, cioè di scene con molteplici modelli in movimento.

Attualmente Horde3D è ancora in fase beta, rilasciata l'ultima volta, come quinta versione, il 31-5-2011. E' al momento disponibile per piattaforme Windows, Linux e MacOSX.

2 Caratteristiche

Andiamo ad analizzare le diverse caratteristiche prevalenti dell'engine Horde3D, senza scendere troppo nei dettagli, per questi si rimanda piuttosto alla documentazione ufficiale.
Il motore grafico è stato designato per supportare un'ampia gamma di effetti e tecniche di rendering, anche ad un livello piuttosto avanzato in modo da poter incontrare le esigenze dei moderni giochi. Si avvale dell'uso delle librerie grafiche OpenGL, per garantire una maggiore portabilità fra le diverse piattaforme, ed è pesantemente basato sull'utilizzo di shaders (scritti dunque in linguaggio GLSL). Horde3D permette la definizione di *contesti di shaders*, che sono in pratica una coppia costituita da un *vertex shader* ed un *fragment shader*, e vengono utilizzati per controllare esattamente come ogni modello debba essere renderizzato. Sono supportati anche gli *uber-shaders*, ossia shader che permettono l'inclusione di parti di codice da altri file esterni.
Come già detto, una delle caratteristiche di Horde3D è quella di saper gestire in maniera piuttosto efficiente moltitudini di modelli contemporaneamente, in particolare i lor vincoli spaziali (come ad esempio modelli che si muovono sopra ad altri modelli), abilità che deriva in buona parte dal particolare modo in cui vengono gestiti tutti gli elementi in scena. Infatti modelli geometrici, sorgenti luminose, sorgenti di particelle e perfino la telecamera di gioco vengono considerati come *nodi* di una struttura gerarchica ad albero, nella quale ciascun nodo figlio dipende direttamente dal padre nel determinare la propria posizione nel mondo. Il movimento di un nodo è in genere il suo spostamento nello spazio relativo rispetto al padre, non nello spazio mondo assoluto, e coinvolge nell'azione anche lo spostamento di tutti i suoi nodi figli. Il processo di rendering della scena si traduce quindi anche in un processo di attraversamento dell'intero albero e nel rendering dei singoli elementi.
Proprio per il fatto che qualunque tipo di elemento è rappresentato da un nodo del grafo di scena, esistono diversi tipi di nodi e ciascuno di essi presenta

proprietà differenti. Nello specifico, un nodo può rappresentare: un modello, una mesh, un *joint* di una struttura scheletrica, una sorgente luminosa, un emettitore di particelle, una telecamera oppure un gruppo di più nodi. Bisogna fare attenzione a non confondere un nodo che rappresenta un modello con uno che rappresenta una mesh, dal momento che quest'ultimo definisce soltanto la struttura geometrica, mentre il secondo l'oggetto stesso nel suo complesso, quindi anche il materiale ed il suo aspetto una volta renderizzato. Di norma un nodo di tipo modello ha come figli uno o più nodi di tipo mesh e fa riferimento ad almeno un sotto-insieme dei loro vertici per definire la propria geometria. Fra i vari nodi citiamo anche i nodi che rappresentano i joint di una struttura ossea, che sono quelli che permettono di effettuare l'animazione scheletrica di un personaggio e l'effetto di *skinning*.

Fra i diversi effetti di rendering supportati, proprio l'animazione dei personaggi una di quelle sulle quali l'engine pone particolare cura. Il sistema di animazione si basa sull'utilizzo di *keyframe* sia per i joint dello scheletro che per le mesh, con ciascun vertice di una mesh in grado di modificare la propria posizione, per la resa dell'effetto di skinning, considerando fino a quattro joint come pesi. Per ciascuna animazione Horde3D utilizza un metodo d'interpolazione *inter-frame* per stabilire la posizione finale di un vertice in un determinato intervallo di tempo. Più animazioni contemporanee sono possibili per ciascun modello ed un meccanismo di *blending layerizzato* (diviso su più livelli) viene adottato per decidere il risultato dell'animazione complessiva. Tale meccanismo pone ciascuna animazione su un proprio livello di priorità ed assegna a ciascuna di esse una percentuale di utilizzo: partendo dal livello di priorità più alto, la posizione di un vertice è calcolata come media pesata del contributo di tutte le animazioni su quello stesso livello, poi, se la percentuale d'utilizzo non supera il valore massimo, si procede con quelle del livello inferiore e così via. Inoltre vengono supportate anche le animazioni facciali tramite i *morph target* dei singoli vertici.

Tutte le informazioni relative alle animazioni di un soggetto sono mantenute da particolari file risorse, con estensione **anim**, codificate in un particolare formato binario proprio dell'engine. In formato binario sono anche i file che descrivono le mesh geometriche (estensione **geo**), i quali racchiudono le informazioni di posizione, coordinate texture, normali, bitangenti e joint collegati

per ciascun vertice. Questi file vengono generati generati tramite un particolare tool fornito assieme all'intero package di Horde3D, il quale genera il file di geometria, il file di animazione (se vi sono animazioni), il file del materiale e il file del sotto-albero della scena che descrive il modello, a partire da un file in formato **Collada** creato con un qualunque programma di modellazione. Questi ultimi due file sono in formato XML e fanno parte dei diversi tipi di file risorse gestiti utilizzati.

Un file relativo al materiale contiene informazioni sui fattori che determinano l'apparenza di un modello, ed in particolare contiene il riferimento agli shader da utilizzare per il suo rendering. Un file di tipo *scene graph* invece descrive una parte dell'intera struttura ad albero della scena, a partire dal nodo radice, e rispecchia le diverse relazioni gerarchiche nel modo in cui i tag XML sono inclusi l'uno nell'altro. A ciascun tipo tag corrisponde dunque un determinato tipo di nodo. Quando questo tipo di file viene generato dal tool precedentemente citato, a partire dal file Collada, esso contiene unicamente un tag per il nodo modello, facente riferimento alla risorsa relativa al materiale, ed un tag per il nodo di tipo mesh come suo unico figlio, facente invece riferimento alla risorsa di geometria. L'uso di questo tipo di file non è a dire il vero necessario, dal momento che è possibile definire un sotto-grafo dell'albero di scena anche in maniera puramente programmatica, ossia definendo passo per passo le relazioni gerarchiche fra i diversi nodi usando le relative chiamate a funzione, ma è senz'altro un ottimo modo per semplificare il lavoro e renderlo maggiormente comprensibile.

Le *risorse* appena citate sono sostanzialmente il contenuto di un file risorsa utilizzato da Horde3D, che viene memorizzato internamente dal sistema e al quale un identificativo univoco, tramite il quale può essere referenziata da nodi dell'albero o da altre risorse. Sono già stati citati i file che contengono informazioni sulla geometria, sulle animazioni, sui materiali e su un sotto-grafo della struttura ad albero della scena, a questi si aggiungono i file degli shader, i file di codice ed i file delle *pipeline*. Inoltre anche i file immagini vengono considerati come file di risorse, ed in particolare vengono identificati come file di risorse di textures.

Una menzione in merito va fatta per i file di shader. A dispetto del nome questi non contengono in realtà solo codice in GLSL, come invece i file di

codice, ma anche tutta una serie di altre informazioni e servono a definire i diversi contesti di shaders utilizzati. All'interno di questi file sono contenute sia parti contenenti vertex shaders e di fragment shaders, divisi dalle apposite sezioni, sia parti che servono appunto per dichiarare i diversi contesti come coppie dei due tipi di shaders. Anche se differenti, più contesti di shaders possono comunque condividere determinate variabili di tipo *uniform*, se dichiarate all'inizio del file.

Infine un file di tipo pipeline descrive appunto una *pipeline* di rendering, ossia un intero processo di rendering che va dalla pulizia dei buffer, al rendering di semplice geometria, al rendering dei modelli con illuminazione e alla visualizzazione di *overlay* bidimensionali in primo piano. Una risorsa pipeline è di norma associata ad un nodo di tipo camera, attraverso il quale viene vista l'intera scena che va renderizzata. E' possibile tramite dividere l'intero processo in diverse fasi e, per ciascuna fase, è possibile stabilire l'ordine delle operazioni che si vuole adottare e decidere quali modelli influenzare. L'utilizzo di questo strumento garantisce grande flessibilità e personalizzazione per l'intero processo di rendering.

Oltre all'effetto d'animazione, Horde3D pone particolare cura anche nell'illuminazione della scena. In particolare due tipi d'illuminazione sono supportate: il *forward lighting* ed il *deferred lighting*. Il primo è assai più semplice e più performante, ma dipende fortemente dal numero di luci in scena, mentre il secondo permette risultati leggermente migliori ma è generalmente più oneroso. Il modo in cui l'illuminazione avviene dipende dal contesto di shader indicato da ciascun nodo relativo ad una sorgente luminosa. Ciascun nodo di questo tipo inoltre fa riferimento anche ad un altro contesto di shader, relativo questa volta alle ombre, che determina appunto come la loro geometria debba essere proiettata. Questo proprio perché Horde3D permette anche un supporto diretto per la gestione delle ombre stesse.

Infine esiste anche una particolare gestione per gli effetti particellari, realizzata attraverso l'utilizzo dei nodi di tipo *emettitori*. Ciascuna sorgente di particelle è in grado di generare particelle ad intervalli più o meno regolari, in direzioni e con velocità prefissate, oppure secondo determinate variabili casuali. Inoltre per ciascuna particella è possibile definire l'equazione di moto e tempo di vita media ed è anche possibile indicare se la stessa sorgente

debba essere chiusa dopo un certo numero di particelle generate.

Figura C.1: Esempio di semplice applicazione sviluppata con l'utilizzo dell'engine Horde3D. Da notare la presenza di numerosi modelli dello stesso tipo, gestiti tutti senza troppi rallentamenti.

3 Note di porting

La difficoltà del portare Horde3D su sistemi propri di dispositivi portatili, quali Android, è stata essenzialmente nelle diverse librerie grafiche supportate, ossia le OpenGL ES anziché le OpenGL classiche, le quali offrono solo un sottoinsieme di API rispetto a quelle originali. E' stato allora necessario modificare o rimuovere molte delle funzionalità originario dell'engine, per adattarlo al nuovo ambiente. Il risultato finale potrebbe a tutti gli effetti venire considerato come una versione nuova e a parte dello stesso motore grafico.
In ogni caso i cambiamenti più evidenti riguardano:

Figura C.2: Altro esempi di utilizzo dell'engine Horde3D. In questo caso vengono messi a risalto soprattutto gli effetti di illuminazione ed effetti particellari.

- la modifica delle direttive di preprocessore, che permettevano di effettuare uno switch fra le parti di codice compilabili su sistemi Windows e quelli invece compilabili su sistemi Unix, mentre adesso considerano solo sistemi iOS e Android
- la modifica delle funzionalità fornite per mezzo delle estensioni, in considerazione di quelle supportate dalle OpenGL ES
- la modifica del il numero di *renderbuffer* utilizzabili, ridotto da 8 a 2
- l'eliminazione dei formati texture non più supportati
- l'eliminazione dell'uso del *multi-sampling*, non più supportato
- l'eliminazione della lettura da moduli texture, non più supportata
- l'eliminazione dell'utilizzo di textures 3D, non più supportate
- l'eliminazione dell'utilizzo dell'*occlusion culling*, non più supportato
- l'eliminazione dell'utilizzo delle *query*, non più supportate

Per ulteriore scrupolo elenchiamo più nel dettaglio anche tutti i cambiamenti effettuati file per file.

- modifica delle direttive di pre-processore nel file **egMain.h** ed **egMain.cpp**, che permettevano la lettura del codice Windows-dipendente ed Unix-dipendente, e modificate per l'utilizzo con piattaforme iOS e Android
- esclusione delle dichiarazioni di variabili e funzioni OpenGL nei file **utOpenGL.h** ed **utOpenGL.cpp**, al posto dei quali vengono utilizzati direttamente gli header di libreria. Le variabili relative alle estensioni sono mantenute ma vengono inizializzate staticamente e solo quelle necessarie
- modificata la funzione **Renderer::drawRenderables** nel file **Renderer.h**, per via dell'eliminazione della modalità di rendering in wireframe
- modificata la funzione funzione **Renderer::drawDebugView**, nel file **Renderer.h**, per via dell'eliminazione della modalità di rendering in wireframe
- eliminata la classe **CPUTimer** nel file **RendererBase.cpp**, divenuta pressochè inutile dal momento che l'estensione OpenGL per il supporto alle timer query non è più supportata
- modificato *l'enum* **TextureFormat** nel file **egRendererBase.h**: sono stati eliminati tutti i tipi relativi ai formati colore ed al loro posto ne sono stati definiti altri due. Questo anche per il mancato supporto alle texture con formato float
- modificate di conseguenza tutte le funzioni in **egRendererBase.cpp** che facevano riferimento ai tipi i texture
- modificata la funzione **RenderDevice::CreateTexture** nel file **egRendererBase.cpp**, che ora non tiene più conto delle estensioni non supportate
- modificata la funzione **RenderDevice::CreateTexture** nel file **egRendererBase.cpp**, in modo che non si utilizzi textures di tipo **sRGB**
- modificata la funzione **RenderDevice::uploadTextureData** nel file **egRendererBase.cpp**, in modo da utilizzare il tipo short al posto del tipo float, causa motivi di incompatibilità con la resa di texture

sul framebuffer. Inoltre **glTexImage2D** viene utilizzato passando come formato interno lo stesso valore del formato, sempre per motivi di compatibilità con le OpenGL ES che supportano complessivamente un numero minore di formati

- rimossa la funzione **RenderDevice::getTextureData** nel file **egRendererBase.cpp**, dal momento che OpenGL ES non offre funzionalità per la lettura da texture
- modificata la funzione **RenderDevice::createRenderBuffer** nel file **egRendererBase.cpp**, in modo da non offrire più supporto per il multi-sampling
- modificate tutte le funzioni per la gestione dei renderbuffer nel file **egRendererBase.cpp**, per incopatibilità dei funzioni da libreria, in particolare per quel che riguarda le funzione **glDrawBuffer**, **glReadBuffer**, le funzioni e le costanti con estensione **EXT**
- eliminata la funzione **resolveRenderBuffer** nel file **egRendererBase.cpp**, per via del mancato supporto alla chiamata API OpenGL **glBlitBuffer**
- eliminate tutte le funzioni della classe **RenderDevice** nel file **egRenderBase.cpp**, che fanno uso delle occlusion query, dal momento che queste non sono più supportate
- modificata la funzione **RenderDevice::applySampleState** nel file **egRendererBase.cpp**, causa mancato supporto del parametro **GL_TEXTURE_COMPARE_MODE**
- modificata la funzione **RenderDevice::commitStates** nel file **egRendererBase.cpp**, causa mancato supporto all'uso di texture 3D
- modificata la funzione **RenderDevice::drawIndexed** nel file **egRendererBase.cpp**, per implementare la funzione **glDrawRangeElements** tramite **glDrawElements**
- modificata la funzione **TextureResource::loadDDS** e **TextureResource::loadSTBI** dal file egTexture.cpp, in relazione ai nuovi formati texture supportati e per evitare la conversione da **RGBA** a **BGRA**
- eliminati i metodi **TextureResource::mapStream** e **TextureResource::unmapStream** nel file **egTexture**, per il mancato supporto alle lettura da textures

- modificata la funzione **PipelineResource::load** nel file **egPipeline**, in relazione ai formati di texture non più supportati
- modificata la funzione **CameraNode::parsingFunc**, nel file **egCamera.cpp**, in modo da evitare l'utilizzo dell'occlusion-culling
- modificata la funzione **CameraNode::setParameterI**, nel file **egCamera.cpp**, in modo da evitare l'utilizzo dell'occlusion-culling

C'è infine da considerare il diverso linguaggio di shading utilizzato, per cui, nel caso di utilizzo degli shaders d'esempio, messi a disposizione con l'engine stesso, anche i relativi file vanno modificati di conseguenza per far fronte alle funzionalità realmente supportate. Ad esempio il tipo uniform **sampler2Dshadow**, che viene utilizzato per la rese degli effetti di ombre, non è supportato dalla versione ES del linguaggio, ragion per cui va sostituito con un tipo supportato (ad esempio il semplice **sampler2D**) ove possibile.

Appendice D

Creazione ed utilizzo di librerie in Android

Esistono due tipi di librerie in ambiente Android: dinamiche (o *shared*), con estension **.so**, e statiche, con estensione **.a**. Per convenzione, per entrambi i tipi, il nome di un file di libreria è preceduto dal prefisso **lib**. Un file di libreria dinamica può venire utilizzato direttamente a livello applicativo, caricandolo tramite l'opportuno metodo **System.load()**, da codice Java, con argomento il nome del file senza estensione e prefisso. Un file di libreria statico viene invece utilizzato dai file di librerie dinamiche, per mettere a disposizione determinate funzionalità.

A partire dalla versione 1.6 di Android è possibile creare librerie scritte in codice nativo, utilizzando le funzionalità dell'NDK. Una volta creata una cartella di nome **jni** all'interno della folder del progetto, contenente i file sorgenti ed i file di configurazione **Android.mk** ed **Application.mk**, è possibile mettere in atto la generazione tramite il comando da script **ndk-build**.
Nel file **Android.mk** vengono indicati i diversi file di libreria da generare e le modalità con cui questi vanno generati. In particolare va indicato:

- il nome del file
- il tipo di libreria da generare
- gli indirizzi (assoluti o relativi rispetto alla cartella **jni**) delle folder in cui sono contenuti i sorgenti
- l'elenco dei sorgenti stessi

- l'estensione utilizzata per i file sorgenti (.cpp o .cxx)
- le eventuali librerie statiche da utilizzare (generate tramite lo stesso file Android.mk o già esistenti)
- le eventuali librerie dinamiche da utilizzare (generate tramite lo stesso file Android.mk o già esistenti)
- le eventuali librerie di sistema da utilizzare
- eventuali indicazioni sull'architettura
- altre eventuali flag di compilazione

Ulteriori opzioni sono disponibili attraverso l'uso del file Application.mk, anche se lasua presenza non è necessaria ai fini della generazione della libreria.

Per un maggiore approfondimento sulla sintassi e sulla semantica dei suddetti file si faccia riferimento alla documentazione ufficiale.

Appendice E

Test di dispositivo Android tramite benchmark

Quanto riferito in questo capitolo si riferisce al lavoro svolto per un precedente progetto, che prevedeva lo sviluppo di un'applicazione di tipo *benchmark* per un dispositivo Android. Come la tipologia del programma stesso può suggerire, lo scopo era quello di testare in maniera intensiva le funzionalità e le potenzialità tecniche (quelle grafiche in particolare) di un device Android. Il dispositivo in questione che era stato prescelto era per altro lo stesso dell'attuale progetto di tesi: un modello di smartphone Sony Ericsson Xperia Play, già approfondito in Appendice A.

1 Descrizione dell'applicazione

Il programma di benchmark in questione si presenta come una sorta di museo virtuale in 3D, delimitato da mura, pavimento e soffitto, e nel quale diversi modelli vengono renderizzati. Numero e tipo di modelli vengono decisi dall'utente prima che la scena sia inizializzata, mentre la loro posizione è sempre disposta lungo i lati, ad intervalli fissi di distanza. Ovviamente la scelta sia del numero che del tipo incide significativamente sulle prestazioni, poichè i vari modelli si differenziano anche parecchio fra loro per numero di poligoni che li compongono. Per la precisione si va da un minimo di poco più di 10 mila, per il modello più semplice, fino ad un massimo di oltre 55 mila poligoni per quello più dettagliato. E' inoltre possibile decidere se visualiz-

zare o meno gli interni della stanza ed anche questa scelta ha un certo peso sulle performance, dal momento che questi stessi interni sono interamente texturizzati.
I vincoli per la realizzazione del programma erano di garantire la compatibilità con dispositivi Android di versione 2.2 o superiore (quindi di utilizzare massimo le API di livello 8) e l'utilizzo delle OpenGL ES di versione 2.0. Il rendering è dunque completamente affidato agli shaders. E' stata inoltre richiesta l'adozione di tutti i possibili accorgimenti in grado di garantire un'ottimizzazione complessiva del sistema.
Per ulteriori dettagli aggiungiamo che l'applicazione è di tipo interattivo, poichè di ottenere l'input con l'utente, tramite touch-screen, per lo spostamento e la rotazione della telecamera, e che la scena è renderizzata simulando il modello d'illuminazione di Phong, ossia con un'illuminazione calcolata per ogni singolo fragment (tutto ovviamente a livello di shaders). Viene adottata anche la tecnica del *level of detail* (semplificata *LOD*), per cui modelli più lontani, oltre una determinata distanza, vengono renderizzati usando una mesh meno definita, contenente un numero inferiore di vertici rispetto all'originale. La tecnica è stata applicata in maniera intelligente per evitare l'effetto di *popping*, con brusche variazioni nell'apparenza dei modelli con l'aumentare della distanza. Per un determinato intervallo di tempo infatti le due diverse mesh del modello continuano a coesistere nello stesso spazio, modificando invece la propria trasparenza per potersi interscambiare. In combinazione a questa viene infine effettuato un cambiamento del metodo di illuminazione, che passa dall'uso di un *Phong shading* all'uso di un *Gouraud shading*.
In Fig. E.1, E.2 e E.3 vengono visualizzati esempi di scene dell'applicazione in esecuzione.

2 Risultati del test

Una volta portata su dispositivo, è possibile testare le prestazioni a run-time dell'applicazione di benchmark e valutare quindi le potenzialità del dispositivo stesso. Una prima valutazione viene ovviamente dalla semplice osservazione mentre questa è in esecuzione, per cui è possibile fare una stima approssimativa del frame-rate. Per un giudizio più preciso è però stata aggiunta anche

Figura E.1: applicazione di tipo benchmark in esecuzione su Android. I modelli visualizzati rappresentano il tipo più complesso fra quelli disponibili, contenente cioè il maggior numero di poligoni.

un'ulteriore funzionalità, che permette in un qualunque momento l'apertura di una finestra riportante la sua media complessiva, calcolata fin dall'inizio dell'esecuzione.
Dopo varie prove, si può constatare come il dispositivo riesca effettivamente a mantenere buoni risultati per scene non eccessivamente complesse. Nello specifico, anche visuallizando gli interni della stanza e utilizzando solamente il tipo di modello con più poligoni, con non più di 20 modelli si riescono ancora ad ottenere mediamente 20-24 frame per secondo.

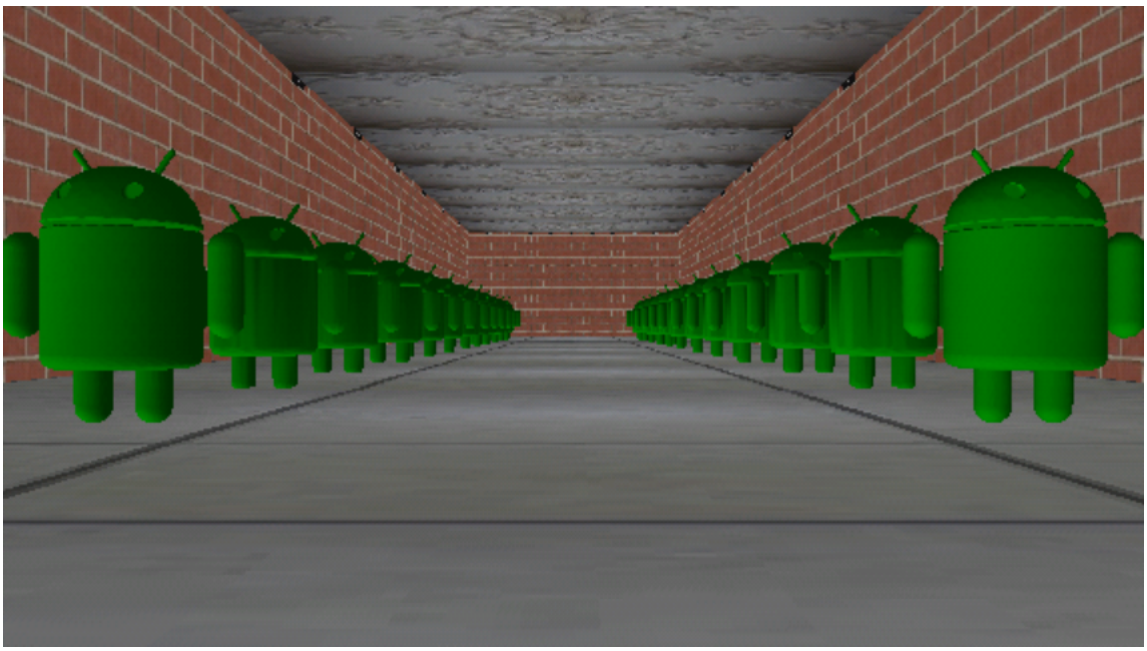

Figura E.2: applicazione di tipo benchmark in esecuzione su Android. I modelli qui visualizzati rappresentano invece il tipo più semplice.

Figura E.3: applicazione di tipo benchmark in esecuzione su Android. In questo caso è piuttosto visibile la differenza fra il modello in primo piano e quello invece dietro, pur essendo dello stesso tipo, anche nella modalità d'illuminazione.

Bibliografia

[1] Kato, H., Billinghurst, M. (1999) *Marker Tracking and HMD Calibration for a video-based Augmented Reality Conferencing System.* I2nd International Workshop on Augmented Reality (IWAR 99). October, San Francisco, USA.

[2] Kato, H., Billinghurst, M., Poupyrev, I., Imamoto, K., Tachibana, K. (2000) *Virtual Object Manipulation on a Table-Top AR Environment.* International Symposium on Augmented Reality, pp.111-119, (ISAR 2000), Munich, Germany.

[3] Kato, H., Billinghurst, M., Morinaga, K., Tachibana, K. (2001) *The Effect of Spatial Cues in Augmented Reality Video Conferencing.* International Conference on Human-Computer Interaction (HCI International 2001), August 5-10th, New Orleans, LA, USA.

[4] Billinghurst, M., Kato, H., Poupyrev, I. (2001) *The MagicBook: A Transitional AR Interface.* Computers and Graphics, November 2001, pp. 745-753.

[5] M. Billinghurst, A. Cheok, S. Prince, H. Kato. *Real World Teleconferencing.* IEEE Computer Graphics and Applications, Nov/Dec 2002, Vol 22, No 6, pp. 11-13.

[6] M. Billinghurst, H. Kato. *Collaborative Augmented Reality.* Communications of the ACM, July 2002, Vol. 45, No. 7, pp. 64-70.

[7] G. Gordan, M. Billinghurst, M. Bell, J. Woodfill, B. Kowalik, A. Erendi, J. Tilander. (2002) *The Use of Dense Stereo Range Data in Augmented Reality.* IEEE and ACM International Symposium on Mixed and

Augmented Reality (ISMAR 2002) 30 Sept. - 1 Oct., 2002, Darmstadt, Germany, IEEE Press, Los Alamitos, CA, pp. 14-23.

[8] E. Woods, P. Mason, M. Billinghurst. *MagicMouse: an Inexpensive 6-Degree-of-Freedom Mouse*. Proceedings of Graphite 2003, Feb 11th-13th, 2003, Melbourne.

[9] Daniel Wagner and Dieter Schmalstieg *ARToolKitPlus for Pose Tracking on Mobile Devices*

[10] Francesco Alinovi *Game Start: Strumenti per Comprendere i Videogiochi*. Springer.

[11] Eray Molla, Vincent Lepetit *Augmented Reality for Board Games*

[12] Amanda Rosler *Augmented Reality Games on the iPhone*. Bachelor Thesis, Blekinge Institute of Technology, Spring 2009

[13] Ohan Oda, Levi J. Lister, Sean White, Steven Feiner *Developing an Augmented Reality Racing Game*

[14] Daniele Bellavista *Game Engine Framework*. Università di Bologna, Seconda Facoltà di Ingegneria Informatica Magistrale , 10 Dec. 2011

[15] Jurgen Stauder *Illumination Analysis for Synthetic/Natural Hybrid Image Sequence Generation*.

[16] Jurgen Stauder *Augmented Reality with Automatic Illumination Control Incorporating Ellipsoidal Models*.

[17] Jurgen Stauder *Estimation of Point Light Source Parameters for Object Based Code*.

[18] Simon Gibson *Illumination Capture and Rendering for Augmented Reality*. RFIA 2004.

[19] Werner Hartmann, Jurgen Zauner, Michael Haller, Thomas Luckeneder, Wolfram Woess *Shadow Catcher: A Vision Based Illumination Condition Sensor using ARToolKit*.

[20] http://www.wikipedia.org

[21] http://www.horde3d.org

[22] http://www.hitl.washington.edu/artoolkit