ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Seconda Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

PROGETTAZIONE E SVILUPPO DI VIDEOGIOCHI PER SMARTPHONE: UN CASO DI STUDIO SU IPHONE

Elaborata nel corso di: Sistemi Operativi LA

Tesi di Laurea di:
ENRICO QUARANTINI

Relatore:
Prof. ALESSANDRO RICCI

ANNO ACCADEMICO 2010–2011
SESSIONE III

PAROLE CHIAVE

Progettazione videogiochi

iOS

Game engine

Cocos2D

Alla mia doppia famiglia

Indice

Introduzione

Il dilagare degli smartphone e dei tablet è un fenomeno sotto gli occhi di tutti. Oggi il mercato degli smartphones sta vivendo il proprio momento d'oro. Questi dispositivi grazie ai loro market per l'acquisto di applicazioni hanno dato il via ad un vero e proprio business da alcuni definito app economy. L'estrema diffusione degli smartphones ha catturato l'attenzione di tantissimi sviluppatori. Gli smartphones integrando diversi tipi di sensori permettono nuovi livelli di interazione da parte dell'utente. Le applicazioni ludiche sono sicuramente tra le principali a beneficiarne.

Obiettivo della tesi è quello di analizzare quali sono le innovazioni introdotte dagli smartphones nel mondo dei videogiochi. Successivamente verranno approfondite quali sono le problematiche da affrontare nello sviluppo di un gioco per questi dispositivi attraverso la realizzazione di un videogioco per la piattaforma iPhone.

Fin dal momento in cui i telefoni cellulari hanno smesso di essere uno status symbol delle utenze business, sono stati introdotti in tutti i modelli dei piccoli giochi. L'evoluzione tecnologica ha fatto sì che questi videogiochi evolvessero di pari passo con i telefoni cellulari. Negli ultimi anni con l'arrivo degli smartphones è stato possibile realizzare giochi di alta fattura, che fino a poco tempo fa erano pensabili solo per le console dedicate.

Gli smartphones non si sono limitati a portare la qualità grafica dei giochi a livelli impensabili per i telefoni cellulari, ma hanno permesso di introdurre nuove interfacce di gioco. All'interno degli smartphones oggi troviamo diversi sensori, primo fra tutti un ampio schermo touch screen. Questo insieme all'accelerometro ha contribuito alla realizzazione di giochi con interfacce più naturali. Per questi motivi è importante analizzare come i giochi siano cambiati grazie agli smartphones. Studieremo inoltre quali sono le architetture hardware adottate da questi dispositivi, quali sono le piattaforme software più diffuse e che strumenti mettono a disposizione degli

sviluppatori.

La realizzazione di un videogioco coinvolge diverse figure: progettisti, programmatori, artisti e game designers. Analizzeremo quali sono i compiti e le responsabilità di ciascuna di queste figure.
Successivamente verranno approfonditi gli aspetti architetturali tipici della normale progettazione di un videogioco. Ci soffermeremo sul game loop: una continua sequenza di operazioni che permette di gestire il flusso di un videogames. Vedremo inoltre come organizzare al meglio la logica di gioco.

La progettazione e la realizzazione di un videogioco sono operazioni molto complesse e richiedono un grosso dispendio di risorse. I game engines permettono di guadagnare molto tempo, risparmiando parecchio lavoro ai progettisti e ai programmatori. Osserveremo come i game engines racchiudano già alcune parti dell'architettura tipica di un videogioco.
Analizzeremo poi alcuni dei game engines più validi per la realizzazione di giochi per smartphones e non solo. Per ciascuno di essi saranno individuati i punti di forza e i punti deboli.

Infine analizzeremo concretamente quali sono le problematiche da affrontare nella realizzazione di un gioco. Sarà infatti realizzato un videogioco per iPhone. Con l'intento di sfruttare i nuovi tipi di input offerti dagli smartphones, ideeremo un semplice gioco che faccia uso dell'accelerometro e del touch screen. Per velocizzare lo sviluppo utilizzeremo il game engine Cocos2D. Studieremo le API che l'engine mette a disposizione degli sviluppatori, inoltre grazie ai sorgenti aperti vedremo anche com'è strutturato.

Una volta appreso il funzionamento del game engine, ci soffermeremo sulla progettazione e l'implementazione dei principali componenti che costituiscono il nostro gioco. Analizzeremo come avviene lo scambio di messaggi tra le diverse entità e studieremo una soluzione che garantisca flessibilità per eventuali future espansioni del gioco.

Capitolo 1

Evoluzione dei videogames su dispositivi mobili

In questo capitolo si fornisce una breve storia dell'evoluzione dei videogames su dispotivi mobili partendo dalle prime *console portatili* fino ai telefoni cellulari di ultima generazione, gli *smartphones*. Segue poi un'analisi delle potenzialità e delle innovazioni introdotte nei videogame grazie alle nuove tecnologie offerte dagli *smartphones*.

1.1 Storia dei videogames su dispositivi mobili

Nei primi anni settanta iniziarono a comparire nei bar i primi *giochi elettronici*, si trattava di grossi cabinati in grado di eseguire il solo gioco per cui erano stati progettati. Richiamarono da subito una certa attenzione e con il passare degli anni si fece sempre più largo la domanda di giochi elettronici portatili, così nella seconda metà del decennio uscirono i primi *tabletop*.

I tabletop riproducevano i grandi successi dei cabinati ma in una versione portatile, così che il giocatore potesse portare sempre con se il proprio gioco preferito, ancora una volta però non si poteva giocare a diversi giochi con lo stesso dispositivo.

Al termine degli anni settanta e in particolare negli anni ottanta il mercato dei videogiochi portatili fece un grosso passo avanti con l'introduzione delle *console portatili*. Le console a differenza dei tabletop permettevano di

giocare a diversi giochi con lo stesso dispositivo che poteva essere programmato tramite *cartucce*[16]. Il primo fra tutti fu il *Microvision* nel 1979, però a causa di una serie di difetti non riscosse grande successo, dieci anni più tardi, nel 1989 entrò in commercio una delle console portatili più vendute di sempre il *Game Boy*.

A metà degli anni novanta i *telefoni cellulari* smisero di essere uno status symbol, con la loro enorme diffusione i produttori iniziarono a rivolgersi ad un bacino di utenti molto più ampio. Anche i videogiochi cominciarono a farsi spazio all'interno delle varie funzioni offerte dai telefoni. I produttori inserivano piccoli giochi all'interno dei telefoni, il loro scopo non era però attrarre i videogiocatori più assidui, bensì offrire semplicemente un piccolo svago, vennero così rispolverati vecchi classici come *Snake*. I cellulari continuarono a diffondersi anche tra i teenager i quali dimostrarono un grande interesse verso i videogiochi al loro interno, la casa produttrice Nokia decise così di rilasciare un cellulare dedicato ai videogiocatori, nel 2003 uscì il *Nokia N-Gage*. Questo cellulare non era più un telefono con all'interno dei giochi, era una vera e propria console portatile a cartucce. Nonostante i giochi fossero di fattura paragonabile alle console portatili di riferimento, in quel momento la prima versione del N-Gage non riscosse molto successo. Nokia presentò qualche anno dopo una versione aggiornata chiamandola *N-Gage QD* che permetteva anche il gioco in multiplayer ma anche questa versione non attrasse molto pubblico e qualche anno più tardi nel 2008 smisero di pubblicare videogiochi.

Contemporaneamente su alcuni modelli di cellulari cominciava a farsi spazio la piattaforma *Java Micro Edition* che metteva a disposizione una serie di librerie per sviluppare applicazioni e videogiochi per dispositivi mobile. Gli utenti potevano così ampliare le funzioni del proprio cellulare scaricando e installando nuovi programmi. Acquistare e installare un'applicazione non era però un'operazione facile e rapida. Le opzioni solitamente erano due: scaricare l'applicazione direttamente dal cellulare oppure scaricarla tramite il computer e trasferirla al telefono tramite cavo o tramite bluetooth. Il download direttamente dal telefono poteva sì esser comodo ma spesso e volentieri risultava un'operazione alquanto costosa, i telefoni erano quasi tutti privi di antenna wifi e i piani internet non erano ancora così diffusi. Ogni operatore telefonico aveva il proprio store online dove vendeva applicazioni e i rispettivi costi venivano direttamente addebitati sulla bolletta telefonica.

Nokia tramite il sistema operativo *Symbian* fu una delle prime aziende a puntare sul mercato degli *smartphones*: dispositivi portatili che abbinano funzionalità di telefono cellulare a quelle di gestione di dati personali. Per certi versi si possono individuare gli smartphones come una via di mezzo tra un telefono cellulare e un *PDA*. Gli smartphones che montavano la piattaforma Symbian riscossero parecchio successo, tuttavia la procedura per installare nuove applicazioni e contenuti continuava ad essere macchinosa in quanto simile a ciò che avveniva per la piattaforma Java ME.

Nel 2007 *Apple* rivoluzionò il mercato lanciando *iPhone*, un telefono rivoluzionario che grazie al display *multitouch* aveva detto definitivamente addio alla tradizionale tastiera. Inizialmente forse nemmeno Apple aveva capito le potenzialità che poteva avere il proprio dispotivo. Durante la presentazione mise subito in chiaro che non sarebbe stato possibile realizzare applicazioni native per il dispositivo ma solamente webapp basate su tecnologia AJAX.

Solamente un anno più tardi la situazione cambiò nuovamente rivoluzionando definitivamente il mercato delle applicazioni per cellulari. Apple accontentò le tante richieste giunte da parte degli sviluppatori rilasciando *iOS SDK* e lanciando l'*App Store* dove qualunque utente con pochi clic, o meglio tap, poteva acquistare e installare centinaia di app direttamente dal proprio iPhone. Tuttavia non fece mancare rigide regole di controllo su tutte le applicazioni, infatti esse prima di essere pubblicate sull'app store vengono provate direttamente da Apple, che si riserva la possibilità di rifiutare applicazioni che non si attengano a determinati standard. Nonostante queste regole l'app store ha attratto immediatamente una grande quantità di sviluppatori e ogni giorno vengono approvate centinaia se non migliaia di nuove app. L'iPhone fu da subito un grandissimo successo commerciale. Viste i milioni di unità vendute e la fascia di utenti davvero ampia sono nate app e giochi per tutte le esigenze e tutti i gusti. L'ampio display multitouch e i vari i sensori inseriti nel dispositivo, primo fra tutti l'accelerometro hanno reso possibile la realizzazione di giochi innovativi e più intuitivi, riuscendo a conquistare utenti che finora non si erano interessati a questo tipo di intrattenimento.

Reduce del successo ottenuto entrando nel mondo dei cellulari Apple decise di affacciarsi anche sul mercato dei *tablet* con *iPad*. Reduce dal consenso ottenuto col sistema operativo *iOS* installato su ogni iPhone, Apple decise di dotare anche il loro tablet dello stesso sistema operativo, mante-

nendo così compatibili quasi tutte le app sviluppate per iPhone. Questa scelta fu largamente criticata al momento della presentazione, tuttavia si rivelò una scelta vincente e il tablet riscosse un gran successo.

Con un po' di ritardo rispetto ad Apple anche *Google* si è mobilitata per entrare nel fiorente mercato degli smartphones e successivamente anche dei tablet lanciando il proprio sistema operativo *Android*. Android nasce come un progetto *open source* e a differenza di Apple non è strettamente legato ad un unico dispositivo fisico. Grazie ad una serie di accordi commerciali con aziende leader nel mercato della telefonia, tra cui *Samsung*, *Htc* e *Motorola* Google è riuscita alla svelta a guadagnare una buona fascia di mercato e attualmente insieme ad Apple si divide buona parte del mercato degli smarphones. I due oltre ad essere divisi da una grande battaglia a livello commerciale sono divisi anche a livello ideologico, da un lato Apple con il suo sistema "chiuso" e i suoi telefoni costosi dall'altro Google e la sua idea di "libertà", "personalizzazione" e una gamma di dispositivi più vasta anche a livello economico. Pubblicare un'app sull'*Android Market* è un'operazione più semplice e immediata, non c'è infatti bisogno di alcuna approvazione e inoltre agli sviluppatori viene lasciata più libertà sui device. Nonostante questo, Apple ad oggi ha raccolto un maggior numero di sviluppatori, complice forse il diverso target di utenti che si affaccia sulle due piattaforme. Gli utenti iPhone sono più disposti a pagare per un'applicazione, mentre quelli Android preferiscono applicazioni gratuite anche se all'interno presentano pubblicità.

Come spesso capita nel campo dell'informatica le evoluzioni sono molto rapide e non tutti riescono a stare al passo delle nuove esigenze. Il mercato della telefonia è stato rivoluzionato con l'avvento degli smarphones e alcuni storici produttori non sono riusciti a stare al passo coi tempi. Produttori come *Nokia* o *RIM* con i suoi *Blackberry* non sono riusciti ad adattarsi rapidamente alle nuove esigenze perdendo così grosse quote di mercato. Anche grossi colossi dell'informatica come *Microsoft* e *Palm* che in passato si erano spartite il mercato dei *PDA* non sono riuscite a mantenere i loro clienti in questa evoluzione. Nel 2010 Microsoft ha iniziato ritagliarsi nuovamente un sua fetta di mercato rilanciando la propria piattaforma mobile *Windows Phone* e stringendo grossi accordi commerciali con Nokia.

1.2 Le innovazioni introdotte nel mondo dei videogames grazie agli smartphones

Gli *smartphones* hanno cambiato drasticamente il panorama dei videogiochi per dispositivi mobile dando vita ad una nuova generazioni di giochi e giocatori. Grazie al vasto numero di sensori che si possono trovare all'interno dei comuni smartphones di oggi i giochi possono spingersi al di là delle tradizionali piattaforme di gaming mobile. L'ubiquità, la connettività, l'integrazione con i propri dati, l'interfaccia innovativa e la popolarità fanno degli smartphones una delle piattaforme più interessanti per lo sviluppo di giochi al giorno d'oggi[4].

Gli smartphones sono prima di tutto telefoni, quindi al giorno d'oggi è naturale portarli sempre con sé ovunque ci si sposti. Con un solo oggetto è quasi possibile rimpiazzare buona parte dei gadget elettronici con cui eravamo abituati ad uscire solo qualche anno fa: lettori mp3, console, fotocamera. In qualunque momento morto della giornata l'utente può estrarre il proprio telefono e mettersi a giocare al proprio gioco preferito oppure acquistarne uno nuovo e usufruirne immediatamente. Grazie allo smartphone sempre a portata di mano gli sviluppatori possono realizzare sia giochi veloci che prevedano partite rapide, cosiddetti *pick up and play* che giochi più complessi che richiedano più tempo da dedicargli. I giochi in cui le partite non durano più di un paio di minuti vanno infatti per la maggiore, ma c'è comunque anche un grande interesse per i giochi più complessi che richiedono più tempo. L'enorme diffusione di questi dispositivi e la loro estrema portabilità fa si che i giochi e le app si diffondano anche tramite il passaparola tra amici. Mostrare i propri giochi preferiti è un attimo e acquistarne e installarne uno nuovo è questione di pochi secondi o al massimo un paio di minuti.

Ci sono diversi motivi per cui una persona può decidere di acquistare uno smartphone, molto spesso i videogiochi non sono fra questi motivi, ma anche chi non si è mai avvicinato a questo intrattenimento trova l'app store e le sua migliaia di giochi molto attraenti. La facilità con cui è possibile acquistare o scaricare un'app, la disponibilità di tantissimi videogiochi e l'innovativa interfaccia possono trasformare chiunque in un giocatore in qualunque momento. Gli sviluppatori così si sono aperti a nuovi target di utenti che fino ad ora erano rimasti fuori da questo mondo.

L'interfaccia utente degli smartphones composta da tutti i suoi *sensori* ha fatto sì che gli sviluppatori potessero realizzare *interfacce* più intuitive

e naturali per i propri giochi. Grazie al multitouch, al microfono e all'accelerometro è stato possibile rendere i controlli quasi invisibili creando un'esperienza di gioco che sia allo stesso tempo coinvolgente e facile da capire. Il giocatore può manipolare direttamente gli oggetti all'interno del gioco toccandoli con il dito o inclinando il telefono nella giusta direzione. Tutto questo ha permesso di dire addio all'antica interfaccia *D-Pad* (il classico controller con le quattro frecce direzionali e una serie di pulsanti). Il giocatore non deve più memorizzare quali azioni corrispondono ad una determinata serie di pulsanti perché l'interfaccia del gioco a volte è il gioco stesso, l'interfaccia viene resa invisibile all'utente. Si può così parlare di *transparent interface*.

Grazie al dilagare delle reti wireless e dei piani telefonici che includono traffico dati, gli sviluppatori possono contare sulla pressoché continua *connettività* da parte dei giocatori. Questo permette di implementare aspetti social all'interno del videogame: l'integrazione con i principali *social network* come *facebook* o *twitter* o l'aggiornamento online di una tabella con i punteggi migliori. Grazie alla rete i giocatori possono sfidarsi direttamente anche se si trovano su due lati opposti del mondo, o in alternativa quando la connessione internet non fosse disponibile due giocatori nelle vicinanze possono connettersi tramite *bluetooth* e sfidarsi. Infine grazie al sistema di *notifiche* un giocatore può ricevere aggiornamenti riguardo lo stato dei suoi punteggi o notifiche di sfide ricevute da parte di altri utenti anche quando il gioco è chiuso.

Essere costantemente connessi non solo può migliorare i giochi ma li rende anche disponibili in qualunque momento e ovunque ci si trovi. Grazie agli app store ciascun gioco è a pochi tap di distanza.

Gli smartphones racchiudono in unico dispositivo una grande quantità di informazioni personali, come i contatti dei propri amici, le foto scattate o la propria libreria musicale. Tutte queste informazioni possono essere sfruttate per personalizzare l'esperienza di gioco. I contatti possono essere utilizzati per vedere se qualche altro amico gioca allo stesso gioco ed eventualmente se è possibile sfidarlo. Allo stesso tempo il giocatore potrebbe voler scegliere un proprio brano preferito tra quelli caricati sul telefono e personalizzare a proprio piacimento la colonna sonora del gioco a cui sta giocando. Infine alcuni giochi offrono la possibilità di sostituire il volto dei personaggi protagonisti del gioco con il nostro volto o quello dei nostri amici, andando a recuperare le immagini dalle nostra galleria foto o scattandone

una direttamente.

1.3 Architettura hardware

Le piattaforme preferite dagli sviluppatori di videogames per smartphone sono iOS , Android e Windows Phone. Attualmente queste tre piattaforme sono quelle di riferimento e si spartiscono buona parte del mercato. L'architettura di riferimento per questi sistemi è l'*architettura ARM*. Tutti i dispositivi iPhone e iPad montano fino dai primi modelli processori basati su questa architettura, la stessa cosa vale per gli smartphone e i tablet di riferimento per il mondo Android e Windows Phone. ARM non produce direttamente i chip, si limita a vendere licenze, cioè proprietà intellettuali. Le licenze permettono di usare i *core ARM* come componenti base attorno ai quali poi costruire uno specifico circuito integrato. Tipicamente su un chip di uno smartphone compaiono oltre alla *CPU* anche memorie e *GPU*. Le caratteristiche cardine dell'archtuttra ARM: semplice, studiata per il *basso consumo* e fortemente *modulare*; fanno sì che venga scelta da quasi tutti i produttori di smartphone e tablet. In particolare l'ultima versione: *ARMv7* e il suo set di istruzioni sono stati scelti da Apple, Htc, Nokia, Samsung e molti altri per i propri dispositivi. Per questi motivi si è deciso di analizzare nel dettaglio questa architettura.

1.3.1 Architettura ARM

L'architettura ARM è un'architettura *RISC* a *32 bit*, molto semplice, studiata per un basso consumo. La sigla RISC significa *Reduced Instruction Set Computer*, questo vuol dire che nella fase di progettazione si è optato per un'architettura semplice e lineare, in grado di eseguire un set di istruzioni limitato ma in tempi inferiori rispetto ad una classica architettura CISC. La maggior parte delle istruzioni vengono eseguite con un solo ciclo di clock. L'architettura ARM è di tipo *load-store*: le istruzioni che elaborano dati operano solo su contenuti di registri interni alla CPU, le istruzioni che possono accedere alla memoria sono solo letture (load, da memoria a un registro interno della CPU) e scritture (store, da un registro alla memoria). Inoltre è stata progettata prevedendo che il processore possa essere affiancato da dei *coprocessori* allo scopo di poter estendere l'architettura per specifici campi applicativi. I processori ARM dispongono di 37 registri interni a 32 bit, di

cui 31 per uso generale e 6 registri di stato[3]. In ogni momento però sono disponibili al programmatore solo 16 di questi e uno o due registri di stato.

I registri disponibili variano a seconda della modalità di funzionamento in cui si trova il processore. L'architettura ARM prevede 7 diverse *modalità di funzionamento*:

- User, modalità standard sotto la quale girano la maggior parte dei processi.
- IRQ, modalità privilegiata, usata quando riceve un'interruzione con bassa priorità.
- FIQ, modalità privilegiata, usata quando riceve un'interruzione con alta priorità.
- Supervisor, modalità privilegiata, usata con l'esecuzioni di istruzioni di interruzioni software e in caso di reset.
- Abort, modalità privilegiata, usata per gestire errori di accesso alla memoria.
- Undef, modalità privilegiata, usata per gestire istruzioni non definite.
- System, modalità privilegiata, usa gli stessi registri della modalità User.

Come si vede in figura 1.1 a ciascuna di queste modalità il processore rende disponibili un certo set di registri. I 16 registri generali (da R0 a R15) possono sempre essere utilizzati indifferentemente come registri sorgente o destinazione, per qualunque istruzione e in qualunque modalità, per questo motivo il set di istruzione dei processori ARM è un *set ortogonale*. Di questi 16 registri a 32 bit:

- da R0 a R12 sono registri di uso generale
- R13 viene usato come *Stack Pointer (SP)*, anche se l'architettura non ne forza l'impiego
- R14 ha la funzione di *subroutine Link Register (LR)*; ci viene salvato l'indirizzo di ritorno quando viene eseguita un'istruzione di Branch and Link (BL)

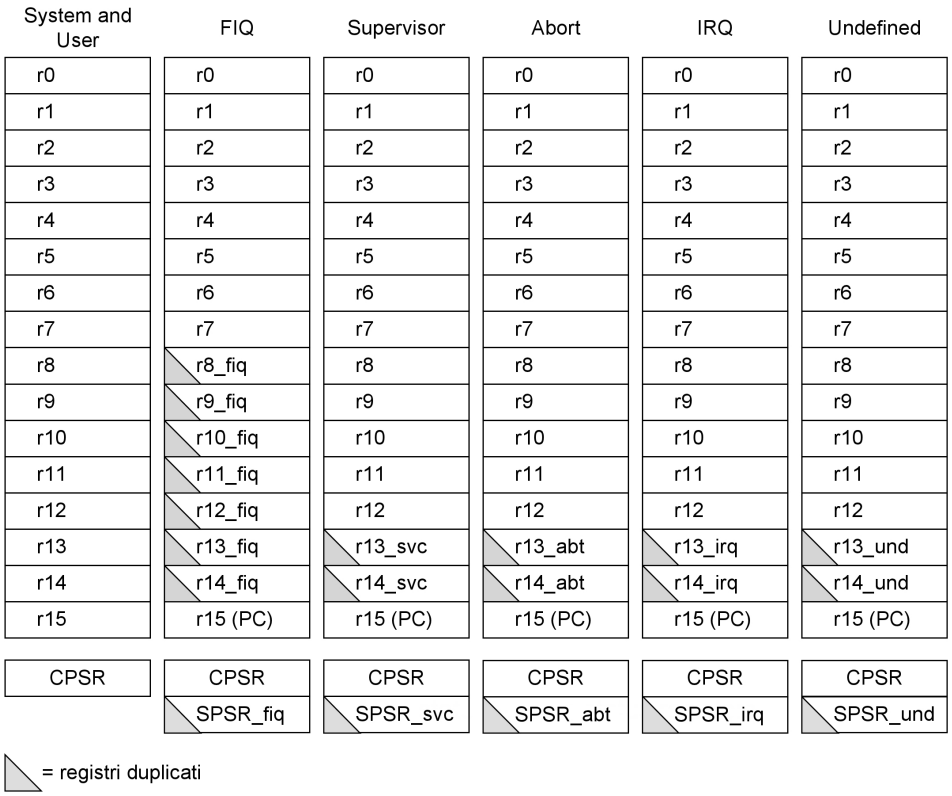

Figura 1.1: Organizzazione dei registri per le varie modalità

- R15 contiene il *Program Counter (PC)*

I registri R13, R14 e R15 possono comunque essere manipolati come registri di uso generale. Alcuni registri sono duplicati (*banked*) e specifici dei modi corrispondenti. Per ciascuna modalità è sempre presente il registro di stato a 32 bit: *Current Program Status Register* (*CPSR*, vedi figura 1.2). Di questi 32 bit:

- 4 bit indicano le 4 condizioni: *Negative*, *Carry*, *Zero* e *Overflow*
- 1 bit indica il set di istruzioni in uso (se ARM o Thumb)
- 2 bit abilitano le interruzioni normali o veloci

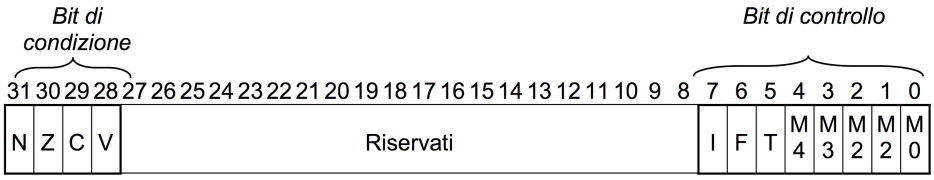

Figura 1.2: Struttura del Current Program Status Register

- 5 bit indicano il modo di funzionamento del processore

I bit restanti sono riservati. Il registro *SPSR* (*Saved Program Status Register*) ha il compito di preservare il valore di CPSR durante la gestione di un'eccezione.

L'architettura ARM prevede 7 diversi tipi di eccezioni, ogniqualvolta il processore incontra un'eccezione cambia il suo stato nelle seguenti modalità:

Tipo eccezione	**Modo**
Reset	SVC
Undefined Instruction	UNDEF
Software Interrupt	SVC
Prefetch Abort	ABORT
Data Abort	ABORT
IRQ	IRQ
FIQ	FIQ

- *Reset*: eccezione generata dal segnale di reset. Consente di inizializzare il processore.
- *Undefined Instruction*: eccezione generata internamente alla CPU quando è stato eseguito il fetch di una codifica di istruzione non valida.
- *Software Interrupt*: eccezione generata dall'apposita istruzione SWI. La routine di servizio assume di norma il ruolo di preparazione a una chiamata a sistema (operativo): il relativo codice viene pertanto eseguito in modalità privilegiata.
- *Prefetch Abort*: eccezione generata dal segnale abort nella fase di (pre)fetch di un'istruzione. Il segnale viene generato nel tentativo di accedere ad un indirizzo di memoria non valido.

- *Data Abort*: come per 'Prefetch Abort' ma in questo caso il segnale abort è generato nella fase di esecuzione un'istruzione.
- *IRQ (Interrupt Request)*: eccezione generata dal segnale IRQ in ingresso alla CPU.
- *FIQ (Fast Interrupt Request)*: eccezione generata dal segnale FIQ in ingresso alla CPU.

A ciascun tipo di eccezione è associato un indirizzo, l'elaborazione dell'eccezione forza il Program Counter ad assumere il valore dell'indirizzo associato, dove è collocata la corrispondente routine di servizio. Quando viene sollevata un'eccezione e inizia la corrispondente elaborazione, vengono eseguite le seguenti azioni:

1. Copia del contenuto del PC (R15) nel registro LR (R14) corrispondente all'eccezione in corso di elaborazione.
2. Copia del registro di stato CPSR nel registro di copia SPSR della corrispondente modalità del processore.
3. Aggiornamento del registro di stato col nuovo modo del processore.
4. Se il tipo di eccezione è Reset o FIQ allora vengono disabilitate le interruzioni veloci settando a 1 il bit F del CPSR.
5. Vengono disabilitate le interruzioni normali, settando a 1 il bit I del CPSR.
6. Aggiorno il Program Counter con l'indirizzo della routine corrispondente all'eccezione sollevata.

Terminata la routine di servizio, il ritorno al codice interrotto ha luogo eseguendo, con un'istruzione atomica, il ripristino di CPSR e PC; ciò può essere attuato utilizzando l'istruzione di caricamento multiplo.

Le prime architetture erano dotate di una *pipeline* a 3 stadi, successivamente è stata portata a 5 stadi per incrementare le prestazioni. Le 5 fasi della pipeline sono: Fetch, Decode, Execute, Memory, Write.

- Instruction Fetch: il processore si occupa di recuperare dalla memoria la prossima istruzione da eseguire.

- Instruction Decode: il processore decodifica l'istruzione e legge gli operandi dai registri.
- Execution: il processore esegue i calcoli veri e propri dell'istruzione.
- Memory: consiste l'accesso ai dati (operandi) dalla memoria nel caso di istruzioni che operano in memoria (come istruzioni di load, o istruzioni con operandi fuori dai registri), oppure se l'istruzione corrente non prevede accesso in memoria lascia transitare l'esecuzione alla fase successiva.
- Write: serve per riporre i risultati dell'operazione all'interno del registro di destinazione.

Al fine di ridurre ulteriormente i cicli di clock i processori ARM implementano al loro interno un *Barrel Shifter* a 32 bit che può essere utilizzato nella fase di esecuzione in contemporanea con la maggior parte delle istruzioni senza penalizzazioni di tempo. Per aumentare ancora nuovamente le prestazioni gli ultimi processori hanno ampliato la pipeline fino a 13 stadi.

Una caratteristica che ha fatto sì che l'architettura ARM si diffondesse così tanto, è sicuramente la possibilità di sfruttare diversi *set di istruzioni*:

- ARM
- Thumb e Thumb 2
- Altre estensioni (es: Jazelle, NEON)

Il *set di istruzioni ARM* è composto da istruzioni a 32 bit con campi di posizione e misura fissa così da facilitare la decodifica. Di questi bit: 4 contengono una condizione settata dall'assemlatore in base all'istruzione. Questa condizione definisce se l'istruzione verrà eseguita o no a seconda dei bit di stato del regsitro CPSR. Questa caratteristica prende il nome di *esecuzione condizionata* ed è uno dei punti forti dell'architettura ARM. Infatti grazie all'esecuzione condizionata è possibile ridurre i salti e gli stalli della pipeline durante l'esecuzione di un programma.
Il set di istruzioni *Thumb* è un set di istruzioni a 16 bit. Il codice Thumb è più leggero ma è dotato di meno funzionalità, è infatti più limitato rispetto al set ARM. Nonostante queste limitazioni, Thumb nel caso di sistemi dotati di limitata larghezza di banda verso la memoria, fornisce prestazioni

migliori rispetto al set di istruzioni completo. Il set *Thumb-2* estende le limitate istruzioni a 16 bit con delle addizionali istruzioni a 32 bit per fornire maggior potenza al processore. La tecnologia Thumb-2 riesce a fornire codice con densità (e quindi occupazione di banda) simile a quello del codice Thumb ma con prestazioni più vicine a quello ARM a 32 bit.
A partire dalla quinta versione dell'architettura è stato introdotto il set di istruzioni *Jazelle DBX* (Direct Bytecode eXecution). Questa tecnologia permette al processore di eseguire nativamente il *Java bytecode*. In realtà il processore non è in grado di eseguire in hardware tutte le 240 istruzioni della Java Virtual Machine, ma soltanto 140 di queste. Le circa 100 istruzioni rimanenti richiedono l'emulazione software prevista da una normale JVM. Tuttavia queste istruzioni sono raramente utilizzate, quindi le prestazioni restano molto interessanti.
Successivamente ARM tornò sui suoi passi eliminando da alcune linee di processori la tecnologia Jazelle DBX e introducendo *ThumbEE*, nota anche come *Jazelle RCT*. Questa nuova tecnologia, basata su Thumb-2, supporta qualunque virtual machine a un costo irrisorio, e allo stesso tempo può eseguire codice nativo, questa volta codice ThumbEE, non più bytecode Java. Il bytecode della virtual machine viene traslato tramite un compilatore JIT in codice ThumbEE, per poi essere eseguito. Risulta evidente che per la conversione è necessario allocare memoria addizionale rispetto a quella in cui risiede il solo bytecode. Grazie alle nuove istruzione e a un controllo sul registro usato per gli indirizzi è possibile controllare automaticamente i puntatori nulli prima di ogni load o store ed eventualmente passare il controllo ad un opportuno handler.
Il processore Cortex-A8 è il primo a integrare la tecnologia *NEON*, un'estensione al set di istruzioni ARM. Questa tecnologia va ad aggiornare la parte di architettura dedicata *SIMD* (*Single Instruction Multiple Data*), che permette di velocizzare l'elaborazione di dati dello stesso tipo. Con questo aggiornamento sono stati aggiunti 32 registri a 64 bit che permettono di lavorare in maniera indipendente con valori in virgola mobile a 16, 32 o 64 bit[3].

Un altro punto di grande forza dell'architettura ARM è la possibilità di estendere le sue funzionalità grazie ai *coprocessori*. I coprocessori permettono di estendere il repertorio di istruzioni. Il repertorio di istruzioni del coprocessore è un repertorio disgiunto rispetto a quello della CPU. In presenza di un'istruzione eseguibile da un coprocessore si instaura un mec-

canismo che porta il coprocessore ad eseguire l'istruzione. Tra il processore e i coprocessori esiste perciò un *protocollo d'interazione* che permette il trasferimento di istruzioni e dati. Anche i coprocessori devono essere basati su un'architettura load-store. Quando in fase di decodifica il processore incontra un'istruzione del coprocessore gli richiede l'esecuzione tramite un segnale, se il coprocessore non è presente si genera un'eccezione. Esistono 3 diverse classi di istruzioni per i coprocessori:

- Istruzioni di elaborazione dati: sono operazioni eseguite direttamente dal coprocessore, quanto la CPU incontra una di queste istruzioni esegue il protocollo di handshake per verificare che uno dei coprocessori presenti accetti l'istruzione.
- Istruzioni load-store: trasferiscono dati dalla memoria ai registri del coprocessore. La CPU si occupa di calcolare l'indirizzo di memoria e di porlo sul bus indirizzi, toccherà poi al coprocessore completare il trasferimento.
- Istruzioni con trasferimento di registri: sono istruzioni che trasferiscono dati tra registri del coprocessore e i registri della CPU.

Tutte queste specifiche fanno sì che l'architettura ARM grazie all'ottimo rapporto consumo prestazioni sia il cuore della maggior parte degli smartphones e tablet oggi in commercio.

1.3.2 I sensori

I videogiochi per cellulari negli ultimi cinque anni hanno fatto passi da gigante grazie agli smartphones. Grazie alla grande quantità di sensori integrati dentro i dispositivi è stato possibile rivoluzionare la normale interfaccia dei videogames[16]. Per questi motivi vale la pena analizzare i sensori integrati nella maggior parte degli smartphones e i tablet di ultima generazione.

Display multi-touch

Dall'introduzione dei display a colori abbiamo assistito ad un continuo ingrandimento degli schermi montati sui cellulari. Con l'avvento della tecnologia multi-touch è stato possibile rimuovere la tastiera e lasciare il pieno

controllo del telefono a questa nuova interfaccia. I display sempre più grandi montati sugli smartphones hanno aperto la strada ai videogiochi, arrivando così a far concorrenza alle console portatili che in alcuni casi hanno display più piccoli degli smartphones oggi in commercio.

La maggior parte degli smartphones e dei tablet oggi in commercio montano un display multi-touch capacitivo che sfrutta la variazione di capacità dielettrica dei condensatori. L'introduzione di questa tecnologia ha permesso l'introduzione di interfacce più naturali all'interno dei videogiochi. Grazie alle *gestures* l'utente può manipolare direttamente gli oggetti che compongono la scena, senza dover usare pulsanti o controller fisici. Grazie a questa tecnologia è inoltre possibile rilevare fino al tocco di 10 dita contemporaneamente, rendendo così possibile anche il multiplayer di più utenti su un unico dispositivo.

Accelerometro

L'accelerometro è un sensore in grado di rilevare e misurare l'accelerazione. Integrato nei tablet e negli smartphones è in grado di rilevare un accelerazione sia lungo il lato verticale che lungo il lato orizzontale del dispositivo. Rilevando l'accelerazione gravitazionale il sistema è in grado di determinare se il telefono si trova in posizione verticale o orizzontale e di conseguenza adattare e orientare le immagini sullo schermo.

Gli accelerometri montati sui tablet e telefoni sono di di tipo capacitivo: sfruttano la variazione della capacità elettrica di un condensatore al variare della distanza tra le sue armature. Un'armatura è fissa col dispositivo, l'altra armatura è dotata di una certa massa e sospesa su un elemento elastico. Un apposito circuito rileva la capacità del condensatore e genera un segnale elettrico proporzionale. Tramite l'accelerometro si può determinare esattamente quanto è inclinato il dispositivo. Grazie alle informazioni fornite da questo sensore è possibile realizzare giochi con interfacce molto più naturali, un esempio eclatante sono i giochi di automobilismo dove il giocatore impugnando il dispositivo con entrambi le mani può sterzare come se stesse tenendo in mano il volante di una vettura.

Giroscopio

Il giroscopio è un sensore in grado di misurare la rotazione rispetto ad un asse. Alcuni smartphones e tablet di ultima generazione vantano tra i vari

sensori anche un giroscopio elettronico a 3 assi, in grado cioè di misurare i gradi di rotazione del dispositivo rispetto ad una terna di assi cartesiani. In realtà i giroscopi elettronici possono misurare direttamente solo la velocità angolare e non i gradi di rotazione, è possibile però arrivare ai gradi della rotazione integrando la velocità angolare.

Esistono già diversi giochi che sfruttano le informazioni fornite da questo sensore ad esempio conoscendo di quanti gradi è stato ruotato il dispositivo e integrando queste informazioni con i dati forniti dall'accelerometro sullo schermo si può simulare l'esplorazione di un ambiente virtuale. L'utente così muovendo il dispositivo può osservare l'ambiente virtuale circostante come se stesse muovendo una finestra che si affaccia su questa realtà artificiale.

Camera

Tutti gli smartphones oggi in commercio e buona parte dei tablet sono dotati di fotocamera con cui si possono scattare foto o girare video. I dispositivi più evoluti ne montano addirittura due: una sul retro del telefono per scattare foto o girare video e una sul fronte del telefono per le videochiamate e gli autoscatti. Attraverso un lavoro di riconoscimento di immagine e ai dati forniti dal GPS è possibile realizzare applicazioni che sfruttino la *realtà aumentata*. Con questo termine si intendono quelle app in grado di mostrare sullo schermo le immagini provenienti in tempo reale dalla fotocamera e sopra ad esse nuovi livelli multimediali contenenti informazioni inerenti a ciò che si sta inquadrando. I giochi che sfruttano questa tecnologia sono molto interessanti in quanto permettono di fondere la realtà virtuale con l'ambiente che circonda il giocatore.

GPS

Il GPS (Global Positioning System) è un sistema in grado di rilevare la posizione di un dispositivo sulla terra. Tramite il collegamento ad almeno 3 satelliti è possibile conoscere la latitudine e longitudine. Quasi tutti gli smartphones e i tablet integrano questo modulo e grazie a queste informazioni e al collegamento internet possono mostrare in pochi istanti la mappa di dove ci si trova. Sono tantissime le applicazioni che sfruttano la posizione del dispositivo, spesso queste informazioni sono integrate con quelle fornite da alti sensori, come il giroscopio o la bussola, per sapere esattamente dove si trova il telefono e come è orientato.

Altri sensori

Gli smartphones integrano molti altri sensori come ad esempio il sensore di prossimità o il sensore di luminosità questi sensori permettono di regolare al meglio la luminosità dello schermo. Le potenzialità di questi sensori non sono ancora state usate all'interno di videogiochi, ma forse un giorno contribuiranno a rendere l'esperienza di gioco più naturale e divertente.

1.4 Piattaforme software

Con l'inizio del nuovo millennio abbiamo assistito ad una rivoluzione nell'ambito dei dispositivi mobili. L'hardware sempre più potente montato sugli smartphones ha permesso la realizzazioni di sistemi operativi mobili sempre più evoluti. In ambito di piattaforme software abbiamo assistito a un notevole progresso. L'evoluzione ha visto inizialmente la grossa diffusione di *Symbian*, un sistema operativo nato completamente per dispositivi mobile. Negli ultimi 5 anni però Symbian ha ceduto il trono a nuovi sistemi operativi: *Android*, *iOS* e *Windows Phone*. Questi nuovi sistemi operativi, a differenza di Symbian, derivano da sitemi desktop, Android è infatti basato sul kernel Linux, iOS sul kernel XNU e Windows Phone sul kernel Windows CE. Segue ora un'analisi di queste tre piattaforme, degli strumenti messi a disposizione agli sviluppatori e della procedura per la pubblicazione delle applicazioni.
I seguenti paragrafi sono basati sulle documentazioni ufficiali fornite da Google[7], Apple[2] e Microsoft[12].

1.4.1 Android

Android nasce dall'esigenza di fornire una *piattaforma aperta* e per quanto possibile standard per la realizzazione di applicazioni in ambito mobile. Android, acquisito da Google nel nel 2005, può essere considerato open in quanto utilizza tecnologie aperte, come il kernel di Linux e in quanto l'intero codice di Android è open source, rilasciato sotto la licenza Apache 2.0. L'obiettivo è sempre stato quello di realizzare una piattaforma open in grado di tenere il passo del mercato senza il peso di royalties che ne possano frenare lo sviluppo[5].

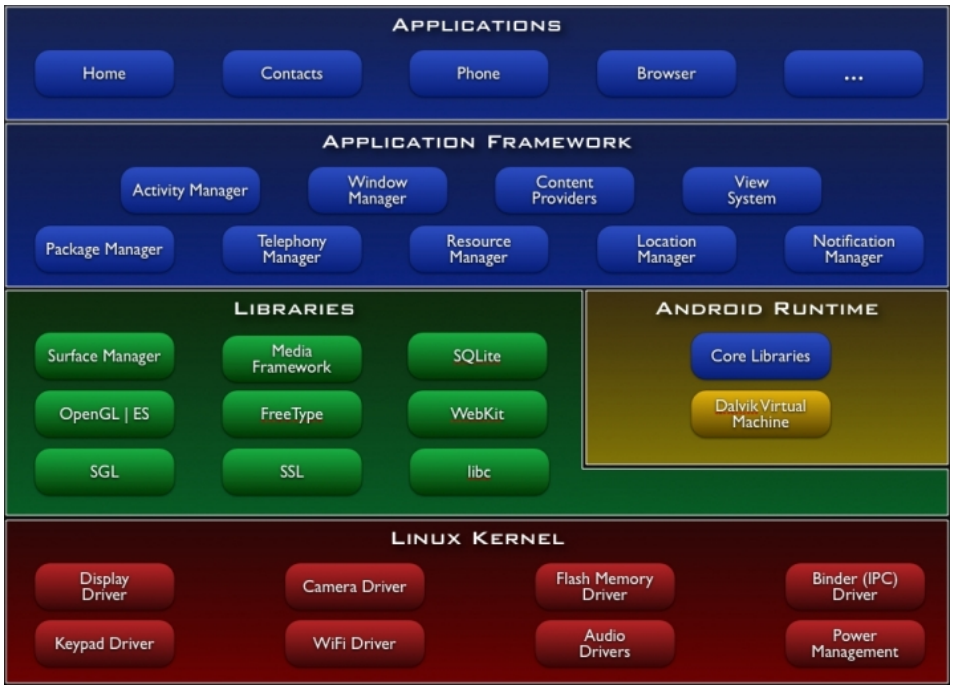

Figura 1.3: Architettura di Android

Android è basato su un'*architettura a livelli*, dove i livelli inferiori offrono servizi ai livelli superiori offrendo così un più alto grado di astrazione (vedi figura 1.3).

Il layer più basso è rappresentato dal *kernel Linux* nella versione 2.6. Questo strato ha il compito di fornire gli strumenti di basso livello per la virtualizzazione dell'hardware, al suo interno sono definiti diversi driver. Oltre ai driver per le varie periferiche possiamo trovare anche la presenza di un driver dedicato alla gestione delle comunicazioni tra processi. Questo driver assume una certa importanza nell'ambiente Android dove ogni applicazione viene eseguita all'interno di un proprio processo. La necessità di avere un kernel che fornisse già tutte le feature di sicurezza, la gestione della memoria, la gestione dei processi e che fosse affidabile e testato ha fatto ricadere la scelta sul kernel Linux. Un produttore intenzionato a installare Android sui propri tablet o smartphones non dovrà far altro che installare il kernel Linux e implementare i driver per il proprio dispositivo.

Sopra al kernel troviamo un livello che contiene un set di *librerie* scritte in C/C++ usate da diversi componenti del sistema. Analizziamo ora ciascun elemento:

- *Surface Manager* ha la responsabilità di gestire le view, ovvero ciò da cui un'interfaccia grafica è composta. Ha il compito di coordinare le diverse finestre che le applicazioni vogliono visualizzare sullo schermo.
- *Open GL ES* sono un versione ridotta e ottimizzata per dispositivi mobile delle librerie grafiche Open GL. Comprendono un vasto numero di API per l'accesso a funzionalità 2D e 3D.
- *SGL* (Scalable Graphics Library) è un'altra serie di librerie, riservate alla grafica 2D. Insieme a Open GL ES compongono il motore grafico di Android. SGL lavora spesso in tandem col Window Manager e il Surface Manager.
- *Media Framework* è un componente in grado di gestire i diversi codec per la riproduzione e registrazioni di immagini, audio e video. Esso è basato sulla libreria open source OpenCore.
- *FreeType* è un motore dedicato al rendering dei font.
- *SQLite* è un compatto DBMS che non necessità di alcuna configurazione. Per operare non utilizza nessun processo separato, ma "vive" nello stesso processo dell'appicazione, per questo motivo si può definire diretto.
- *WebKit* è un browser engine open source.
- *SSL* (Secure Socket Layer) è la libreria che permette lo scambio di messaggi cifrati sulla rete.
- *Libc* è una versione ottimizzata per dispositivi basati su Linux embedded della libreria standard C.

Android non è basato su un nuovo linguaggio, è basato sul *linguaggio Java*. Questo è un grosso vantaggio per gli sviluppatori interessati alla piattaforma, i quali per realizzare le proprie applicazioni non devono imparare un nuovo linguaggio. Inoltre la scelta di un nuovo linguaggio avrebbe costretto Google a realizzare un compilatore e un debugger specifico.

La scelta di Java porta però con se diverse problematiche, il compilatore infatti non fornisce direttamente del codice eseguibile ma del bytecode che per essere eseguito necessita della Java Virtual Machine (JVM). Il problema sorge perché i dispositivi che intendono adottare la *Java Virtual Machine* devono pagare delle royalty alla Sun. Questo entra in contrasto con la natura open di Android[6].
Per risolvere questo problema Google ha realizzato una propria virtual machine chiamata Dalvik. La *Dalvik Virtual Machine* (DVM) è ottimizzata per dispositivi a risorse limitate e come la JVM include il garbage collector per la gestione della memoria. Introducendo questa nuova macchina virtuale il codice Java dopo essere stato compilato in bytecode viene convertito dal suo formato .class ad un formato Dalvik compatibile .dex. Ogni applicazione sarà quindi eseguita in un processo separato con una propria istanza della Dalvik VM.
Android include un set di *Core libraries* costituito dalla maggior parte delle funzionalità fornite dalle librerie standard Java in formato Dalvik eseguibile.

L'*Application Framework* racchiude un'insieme di componenti di alto livello che sfruttano le librerie viste finora. Questo strato mette a disposizione una serie di API e componenti per l'esecuzione di precise funzionalità per tutte le applicazioni Android. Tutte le applicazioni sulla piattaforma Android si interfacciano con questo layer, per questo motivo tutte le applicazioni, originali o di terze parti possono essere rimpiazzate o estese, dando così origine al motto che contraddistingue il sistema: "all applications are equals".
Vediamo ora quali librerie compongono questo strato:

- *Activity Manager* è un componente con la responsibilità di gestire le activity. Una Activity rappresenta una possibile interazione dell'utente con l'applicazione e può essere associato al concetto di schermata.

- *Package Manager* è una serie di strumenti che si occupano della gestione dei pacchetti delle applicazioni, della loro installazione e rimozione.

- *Window Manager* offre funzionalità per la gestione delle finestre delle diverse applicazioni sullo schermo del dispositivo.

- *Telephony Manager* permette l'interazione con le funzionalità offerte da un normale telefono cellulare.

- *Content Provider* è componente con la responsabilità di gestire la condivisione di informazioni tra i vari processi. Offre un funzionamento simili a quello di un repository condiviso.

- *Resource Manager* mette a disposizione una serie di API per gestire l'insieme di file di cui ha bisogno un'applicazione per funzionare. Come ad esempio: immagini o file di configurazione, come avviene per gli eseguibili anche questi file vengono ottimizzati per il loro utilizzo all'interno del dispositivo.

- *View System* è un componente che si occupa del rendering delle interfacce e della gestione degli eventi associati ad esse.

- *Location Manager* mette a disposizione delle API per la realizzazione di applicazioni che sfruttano la localizzazione del dispositivo.

- *Notification Manager* è uno strumento che offre la possibilità alle applicazioni di inviare notifiche al dispositivo, il quale si occuperà di segnalarle all'utente con i meccanismi che conosce.

Il *Software Development Kit* (*SDK*) di Android è disponibile gratuitamente online. Al fine di semplificare lo sviluppo, Google ha realizzato un plug-in per *Eclipse*: uno degli IDE più utilizzato in ambito di programmazione Java. Il plug-in *Android Development Tools* (ADT) mette ha disposizione dei programmatori diversi strumenti tra cui un simulatore che permette di testare le proprie applicazioni su diversi dispositivi (virtuali) con diverse versioni del sistema operativo. Sempre tramite l'ambiente Eclipse gli sviluppatori possono testare liberamente le proprie applicazioni su *dispositivi fisici* con sistema Android. Il testing su dispositivi fisici è molto importante e certe volte è indispensabile in quanto il simulatore non può simulare le funzionalità offerte da diversi sensori come l'accelerometro e molti altri.

Per pubblicare applicazioni sull'*Android Market* è necessario un account sviluppatore. Registrarsi come sviluppatori ha un costo di 25 $ una tantum, anche se si ha intenzione di pubblicare solo applicazione gratuite. Questa quota serve per favorire una maggiore qualità dei prodotti sul Market (ad esempio per avere meno spam). Una volta registrati è possibile pubblicare direttamente le proprie applicazioni. Android Market non prevede infatti alcun processo di approvazione. L'assenza di un processo di approvazione

può portare alla pubblicazioni di applicazioni di scarsa fattura o di applicazioni che violino la privacy, per questo motivo ciascuna applicazione prima di essere installata comunica all'utente i permessi di cui a bisogno per funzionare, l'utente può così decidere di non installare applicazioni le cui richieste sembrano eccessive o non necessarie. Infine gli utenti possono lasciare valutazioni e piccole recensioni per le applicazioni installate in modo da guidare futuri utenti verso certo app piuttosto che altre. Per quanto riguarda le applicazioni a pagamento Google trattiene una commissione del 30% sul prezzo dell'applicazione venduta.

1.4.2 iOS

iOS è il sistema operativo di Apple disponibile esclusivamente per i propri dispositivi: iPhone, iPad e iPod Touch. É basato su un'*architettura a livelli*

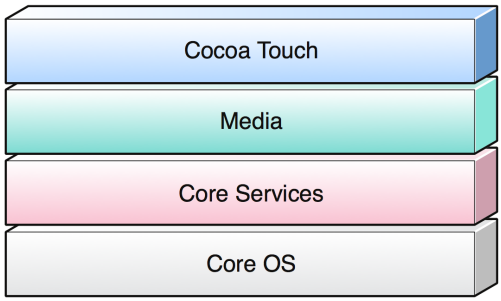

Figura 1.4: Layers di iOS

(vedi figura 1.4), dove i livelli inferiori offrono servizi ai livelli superiori offrendo così un più alto grado di astrazione. Nella realizzazione di un'applicazione generalmente si opera usando i frameworks messi a disposizione dal livello più alto. Questi framework forniscono un'astrazione object-oriented per i costrutti dei livelli inferiori. L'alto livello di astrazione fornito dal livello più alto rende la scrittura del codice più facile e più veloce, ciò nonostante, i frameworks dei livelli inferiori sono comunque a disposizione degli sviluppatori. La maggior parte delle librerie sono scritte in Objective C e C. I due livelli più bassi: Core OS e Core Service e parte del livello Media derivano direttamente dall'architettura del sistema operativo per desktop Mac OS X. Analizzeremo ora le responsabilità e le funzioni di ciascuno strato.

Il livello *Core OS* contiene gli strumenti di basso livello su cui sopra sono costruite la maggior parte delle altre tecnologie. Nella realizzazione di un'applicazione solitamente si accede a questo livello esclusivamente indirettamente tramite i livelli superiori. Nella parte più bassa troviamo il *System* che comprende il *kernel*, i *drivers* e le *interfacce Unix* di basso livello. Il sistema iOS deriva da Mac OS X che è un sistema operativo Unix-based. Come per Mac OS X il kernel scelto è *XNU*, un kernel ibrido basato su l'unione del codice del microkernel Mach e del kernel monolitico FreeBSD è responsabile di ogni aspetto del sistema operativo. Gestisce la memoria virtuale, i threads, il file system, l'accesso alla rete e le comunicazioni tra processi. I drivers forniscono un'interfaccia ai frameworks di sistema per accedere all'hardware vero e proprio. Per questioni di sicurezza l'accesso al kernel e ai drivers è limitato ad un ristretto set di frameworks e applicazioni. iOS mette a disposizione un set di interfacce per accedere a diverse features di basso livello del sistema. Queste interfacce sono raccolte nella libreria *LibSystem*. Tutte le interfacce sono scritte in C e forniscono il supporto per: threads, accesso alla rete, file system, I/O, servizi Bonjour e DNS, allocazione della memoria e computazione matematica.
Il Core OS mette a disposizione i seguenti framework:

- *Accelerate Framework* contiene una serie di interfacce per l'escuzione di DSP, algebra lineare e image-processing. Queste librerie sono ottimizzate per tutte le configurazioni hardware dei dispositivi basati su iOS.

- *Core Bluetooth Framework* permette di interagire con dispositivi Bluetooth LE. Mette a disposizione interfacce per cercare accessori, connettersi o disconnettersi da uno di questi, scambiare attributi con esso e inviare o ricevere notifiche.

- *External Accessory Framework* offre il supporto per la comunicazione con accessori collegati al dispositivo. Gli accessori possono essere collegati attraverso l'attacco dock del dispositivo oppure senza fili tramite Bluetooth. Questo framework permette di ricevere informazioni sull'accessorio collegato e di iniziare una comunicazione con questo. Una volta iniziata la comunicazione è possibile manipolare l'accessorio usando direttamente i comandi che mette a disposizione.

- *Security Framework*. In aggiunta alle caratteristiche intrinseche di sicurezza, iOS fornisce questo framework per garantire la sicurezza dei dati che l'applicazione deve gestire. Questo framework fornisce interfacce per la gestione di: certificati, chiavi pubbliche e chiavi privati. Fornisce inoltre il supporto per la crittografia.

Gli sviluppatori generalmente interagiscono direttamente col livello Core OS nelle situazione in cui è necessario affrontare in modo esplicito questioni di sicurezza o quando devono interagire con un accessorio fisico esterno.

Il livello *Core Services* contiene i servizi fondamentali usati da tutte le applicazioni. Contiene un'ampia serie di frameworks principalmente in object-oriented:

- *Address Book Framework* mette a disposizione una serie di strumenti per accedere, aggiungere, modificare contatti all'interno della rubrica.

- *CFNetwork Framework* contiene interfacce per lavorare ad un alto livello di astrazione con i protocolli di rete. Questo framework torna spesso utile per comunicare (anche in modo sicuro) con server FTP e HTTP o nel risolvere domini DNS.

- *Core Data Framework* offre una tecnologia per la gestione del modello dati delle applicazioni basate sulla struttura Model-View-Controller. Core Data è specifico per quelle applicazioni in cui il modello dati è già altamente strutturato. Come vedremo successivamente è disponibile un tool grafico per disegnare e strutturare il modello dati.

- *Core Foundation Framework* contiene un set di interfacce C che permettono la gestione delle informazioni e dei servizi per le applicazioni iOS. Oltre ad includere il supporto per gli array, i set ed altre strutture dati include anche il supporto per i threads, la gestione delle stringhe e molto altro.

- *Foundation Framework* ingloba in una serie di intefacce Objective-C le funzionalità offerte dal Core Foundation Framework.

- *Core Location Framework* mette a disposizione delle applicazioni le informazioni riguardo la posizione del dispositivo. Il framework fa uso del GPS, del segnale di rete telefonico e del segnale Wi-Fi per localizzare il dispositivo.

- *Event Kit Framework* offre interfacce per accedere al calendario dell'utente. Permette di leggere, modificare e aggiungere eventi.
- *Quick Look Framework* offre un'interfaccia per mostrare il contenuto dei file che la nostra applicazione non è in grado di gestire.
- *Store Kit Framework* mette a disposizione una serie di strumenti per gestire l'acquisto di nuovi contenuti direttamente all'interno dell'applicazione.
- *System Configuration Framework* fornisce le interfacce per determinare la configurazione di rete del dispositivo. Grazie a questo framework l'applicazione può individuare se il dispositivo è connesso tramite Wi-Fi o tramite connessione dati cellulare.

Il livello *Media* contiene le tecnologie per offrire una migliore esperienza multimediale in ambito di elaborazione, creazione e riproduzione di contenuti audio, video e applicazioni grafiche. Queste tecnologie, ottimizzate per i dispositivi mobile, snelliscono la fase di progettazione e realizzazione di un applicazione in questi ambiti. Analizziamo ora i principali framework che lo compongono:

- *Assets Library Framework* permette di accedere a foto e video contenuti nella libreria dell'utente. Oltre a caricare contenuti permette anche di salvarne di nuovi.
- *AV Foundation Framework* è una libreria che permette di registrare e riprodurre file audio e video. Con la versione 5 di iOS integra anche il supporto alla tecnologia AirPlay che estende la riproduzione dei contenuti su dispositivi compatibili (es: AppleTV).
- *Core Audio* è un interfaccia in linguaggio C per l'elaborazione di audio in formato stereo.
- *Core Graphics Framework* contiene l'interfaccia per le API di disegno di Quartz 2D. Quartz è un motore per il disegno usato anche in Mac OS X.
- *Core Image Framework* offre una serie di filtri per manipolare immagini e filmati. Il vantaggio di usare questi filtri è che operano in modo non distruttivo in modo che le immagini originali siano ripristinabili.

- *Core MIDI Framework* offre le funzionalità per comunicare con dispositivi MIDI, come una tastiera o un sintetizzatore connessi al dispositivo mediante la rete o un connettore dock.

- *Core Text Framework* contiene una serie di interfacce C per la disposizione, manipolazione e rendering di testo e font. É un framework destinato a chi richiede elevate capacità di gestione del testo, come ad esempio le applicazioni di word processing.

- *Image I/O Framework* offre interfacce per l'importazione e esportazione di dati e matadati di immagini.

- *Media Player Framework* offre un supporto ad alto livello per la riproduzione di contenuti audio e video da un'applicazione tramite il player standard di sistema.

- *OpenAL Framework* è un interfaccia cross-platform standard per il rendering di audio professionale a tre dimensioni.

- *OpenGL ES Framework* offre diversi strumenti e interfacce per il disegno e l'animazione di contenuti 2D e 3D. É un framework basato sul linguaggio C, lavora a stretto contatto con l'hardware del dispositivo per fornire un frame rate dello schermo elevato, è usato molto nei videogiochi. Dalla versione 3.0, iOS supporta le librerie OpenGL ES 2.0 e 1.1.

- *Quartz Core Framework* contiene le interfacce di Core Animation. Core Animation è un set di librerie per il rendering di animazioni e effetti. É usato da diverse parte di iOS, in particolare dalle classi UIKIt che gestiscono l'interfaccia.

Lo strato *Cocoa Touch* contiene i frameworks chiave per la creazione di applicazioni iOS. Questo strato definisce le infrastrutture di base delle applicazioni e il supporto per tecnologie avanzate come il multitasking, l'input multi-touch, le notifiche push, e molti servizi di alto livello di sistema. Vediamo nel dettaglio i framework che lo compongono e i servizi che offrono:

- *Address Book UI Framework* offre interfacce per offrire all'utente la normale interfaccia di sistema per aggiungere, modificare o selezionare contatti all'interno della rubrica.

- *Event Kit UI Framework* permette di inserire all'interno di un'applicazione la possibilità di modificare o aggiungere eventi al calendario tramite l'interfaccia standard del sistema operativo.

- *Game Kit Framework* consente di aggiungere connessioni di rete peer-to-peer alle applicazioni. In particolare, questo framework fornisce il supporto per il peer-to-peer e la connettività. Anche se queste caratteristiche sono più comunemente si trovano in multiplayer giochi in rete, è possibile incorporarle anche in applicazioni diverse da giochi.

- *iAd Framework* permette di ineserire all'interno delle applicazioni banner e contenuti pubblicitari tramite il programma di inserzioni gestito da Apple.

- *Map Kit Framework* offre funzionalità per inserire all'interno di applicazioni delle mappe navigabili e localizzate in base alla posizione del dispositivo.

- *Message UI Framework* permette di inserire all'interno delle applicazioni la possibilità di creare e inviare email e sms.

- *UIKit Framework* fornisce le classi necessarie per costruire e gestire l'interfaccia utente di un'applicazione iOS. Esso fornisce l'oggetto applicazione, la gestione degli eventi, finestre, viste e controlli appositamente progettati per l'interfaccia touch screen.

Il *Software Development Kit* (*SDK*) di iOS è disponibile esclusivamente per utenti Mac. Sono forniti diversi strumenti per lo sviluppo di applicazioni, tutti racchiusi all'interno dell'IDE *Xcode*. XCode permette di scrivere e compilare codice in linguaggio C e *Objective-C*: il linguaggio principale della piattaforma iOS. Dalla versione di iOS 5.0 è stato aggiunto in fase di compilazione il supporto del *Automatic Reference Counting* (*ARC*) che si occupa della gestione del ciclo di vita degli oggetti Objective-C. Lo sviluppatore non deve più ricordarsi quanto fare una retain o una release su un'oggetto, ARC valuta il tempo di vita degli oggetti e inserisce le chiamate ai metodi appropriati in fase di compilazione. Nonostante le funzioni possano risultare simili, ARC non va confuso con un Garbage Collectore che invece agisce durante la fase di esecuzione dell'applicazione.
Oltre agli editor di codice integra il *Core Data model editor* che permette,

tramite un'interfaccia grafica basata su tabelle, di progettare e gestire modelli dati. Anche per disegnare l'interfaccia grafica è disponibile *Interface Builder*, un tool che tramite l'interfaccia grafica semplifica notevolmente il lavoro degli sviluppatori[9].
Per testare le applicazioni è disponibile un simulatore per ciascuno dispositivo iOS che può essere usato liberamente. Per testare le applicazioni su dispositivi fisici è necessario un account sviluppatore che ha un costo di 79 euro all'anno, una volta registrati è possibile testare le applicazioni su un massimo di 100 dispositivi diversi.

Per pubblicare la propria applicazione sull'App Store è necessario ancora una volta l'account sviluppatore. Le app prima di essere pubblicate vengono però testate, Apple si riserva la possibilità di rifiutare l'applicazione per motivi tecnici o commerciali. Per quanto riguarda le applicazioni a pagamento viene trattenuta una commissione del 30% sul prezzo dell'applicazione venduta.

1.4.3 Windows Phone

Windows Phone è la piattaforma per smartphones di Microsoft. L'ultima versione rilasciata è la 7.5 denominata Mango. Come i due sistemi visti precedentemente anche questa piattaforma è basata su un'*architettura a livelli*.

Il livello più basso è costituito dall'*hardware BSP*. Questo livello contiene tutti i *driver* forniti dal produttore, Windows Phone non è legato ad un particolare produttore. Subito sopra questo strato troviamo il *kernel*, basato sul kernel di *Windows CE 7*. Windows CE è un sistema operativo sviluppato da Microsoft orientato a dispositivi portatili e sistemi embedded, supporta diverse architetture hardware tra cui quella ARM, che sta alla base della maggior parte degli smartphones di oggi. Grazie allo strato hardware BSP il kernel non è direttamente a contatto con l'hardware, diventando così, indipendente. Il livello kernel ha diverse responsabilità, tra queste possiamo individuare: la gestione della memoria virtuale, la gestione della rete e lo scheduling di threads e processi.

Nella scrittura di applicazioni per Windows Phone non è possibile fare uso di *codice nativo*, per questo motivo i livelli visti finora assumono un interesse relativo agli sviluppatori.
Nell'architettura software è presente un livello invalicabile costituito dal *Common Language Runtime* (*CLR*). CLR è una *macchina virtuale* per

la piattaforma *.NET*. Questa piattaforma nata inizialmente per sistemi desktop e server è costituita da una serie di linguaggi object oriented. La macchina virtuale CLR assume un ruolo molto simile a quello che avviene con la Java VM. Il codice scritto viene tradotto tramite un compilatore in *Common Intermediate Language* (*CIL*). Il CIL è un linguaggio assembly orientato agli oggetti, ed è completamente basato su stack. Successivamente in fase di runtime il codice CIL viene tradotto grazie alla macchina virtuale CLR in codice macchina.
Window Phone 7 integra *.NET Compact Framework* 3.7, una versione del framework .NET disegnata per girare su dispositivi con risorse limitate. Questa versione della piattaforma supporta i linguaggi *C#* e *Visual Basic .NET*.

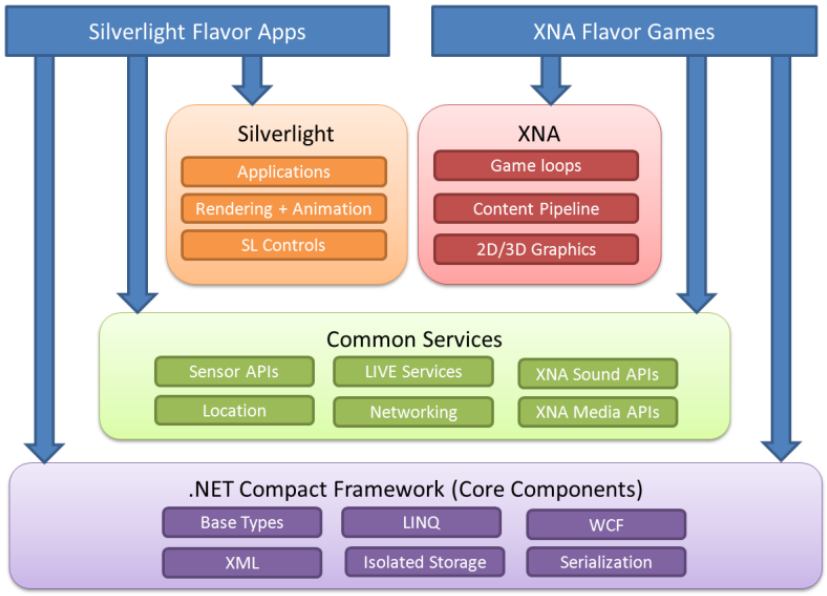

Figura 1.5: Architettura di Windows Phone 7

Come si vede in figura 1.5 lo sviluppo delle applicazioni si basa su due frameworks: *Silverlight* e *XNA*. A partire dalla versione 7.1 di WP è possibile usare entrambe queste tecnologie all'interno della stessa applicazione. Nonostante questa possibilità le due tecnologie sono nate per scopi differenti.

Silverlight è un framework nato per lo sviluppo di applicazioni basate sulla normale interfaccia grafica dei dispositivi Windows Phone. Tramite Silverlight è possibile realizzare velocemente applicazioni che sfruttino eventi o che integrino al loro interno tecnologie web come HTML o Javascript.

XNA offre tecnologie utili nella realizzazione di tutte le applicazioni che manipolano dei contenuti multimediali. Questo framework offre quindi tantissime librerie per l'elaborazione grafica in 2D e in 3D. Offre anche tante altre funzionalità per l'acquisizione e elaborazione di materiale multimediale, come la possibilità di ottenere immagini o video provenienti dalla fotocamera del telefono o ancora registrare audio proveniente dal microfono. XNA non è una tecnologia nuova, queste librerie sono già da diversi anni disponibili per le varie versioni di Windows per i desktop e per la console XBox 360.

Il *Software Development Kit* (*SDK*) di Windows Phone è disponibile esclusivamente per il sistema operativo Windows 7 e Windows Vista. Per lo sviluppo è disponibile gratuitamente l'IDE *Visual Studio Express 2010*, con la quale oltre allo scrivere codice è possibile disegnare interfacce grafiche.
Per quanto riguarda il testing e la pubblicazione delle applicazioni, Microsoft ha optato per una soluzione molto simile a quella scelta da Apple. Per testare le applicazioni è disponibile un simulatore che permette di emulare uno smartphone, tra le varie opzioni del simulatore è presente una funzione che permette di simulare dei movimenti del telefono muovendo il cursore del mouse, ottenendo così dati utili alle applicazioni che fanno uso dell'accelerometro. Per testare le applicazioni su dispositivi fisici è invece necessario un account sviluppatore che ha un costo di 99 $ all'anno, una volta registrati è possibile testare le applicazioni su un numero limitato di dispositivi.
Per pubblicare la propria applicazione sul Marketplace di Windows Phone è necessario ancora una volta l'account sviluppatore. Le applicazioni prima di essere pubblicate vengono testate dalla Microsoft, la quale si riserva la possibilità di rifiutare l'applicazione. Per quanto riguarda le applicazioni a pagamento viene trattenuta una commissione del 30% sul prezzo dell'applicazione venduta. Vi è infine un limite di 100 applicazioni gratuite all'anno, superato questo limite pubblicare una nuova applicazione gratuita costa 20 $.

Capitolo 2

Aspetti dello sviluppo di un videogioco

In questo capitolo si fornisce un'analisi delle problematiche da affrontare nello sviluppo di videogiochi. Nella prima parte verrà presentata un'analisi alla tipica struttura di un team per lo sviluppo di videogiochi, mentre successivamente verrà analizzata l'architettura e alcuni aspetti di programmazione dei videogiochi.
Il materiale sul quale è basato il capitolo è tratto dai libri: Introduction to Game Development [14], Game Coding Complete [11], Game Engine Architecture [8].

2.1 Struttura tipica di un team di sviluppo

Analizziamo ora quali sono le principali figure che compongono un team di sviluppo di videogiochi e quali responsabilità hanno. Possiamo individuare queste principali discipline: progettisti, programmatori, artisti, game designers. Ciascuna di queste discipline può a sua volta essere scomposta in diverse categorie. Approfondiamo ora ciascuno dei ruoli elencati.

Progettisti e programmatori

I progettisti hanno il compito di progettare e sviluppare la parte software che compone il gioco e gli strumenti per semplificare e velocizzare lo sviluppo. I programmatori, contribuiscono alla stesura del codice e possono essere divisi

in due gruppi, a seconda dell'ambito a cui si dedicano: *runtine programmers* (coloro che lavorano sul motore e sul gioco stesso) e *tools programmers* (i quali lavorano alla realizzazione di strumenti che permettono al resto del team di lavorare più efficacemente). Dentro queste aree possiamo trovare progettisti e programmatori dedicati a parti ben precise come: rendering engine, intelligenza artificiale, sistema per la gestione di collisioni, physics engine e molte altre ancora.

Progettisti con una grande esperienza possono ricoprire il ruolo di leadership nell'ambito tecnico. Essi spesso continuano a collaborare alla progettazione e stesura del codice ma si occupano anche: di individuare e gestire le scadenze del team e di prendere decisione sulle direzioni tecniche del progetto.

Artisti

Gli *artisti* si occupano di produrre tutti i contenuti visuali e audio di un gioco. Partendo da immagini e schizzi presentano al team, nelle prime fasi del progetto, un *concept*, cioè una visione di quello che sarà il lavoro finito. Se il gioco in lavorazione è in tre dimensioni, una parte degli artisti si dedicherà alla produzione di *modelli 3D*. Questi modelli, necessari per ciascun oggetto che compone il gioco, sono composti da geometrie tridimensionali. I *texture* artists si dedicheranno al disegno di immagini 2D da applicare sui modelli 3D per fornire i dettagli di ciascun oggetto. Un filone speciale degli artisti si dedica alle animazione dei personaggi e degli oggetti, oltre ad avere conoscenze dedicate a tale ambito gli artisti dedicati a questo compito devono conoscere anche il funzionamento del game engine che sta dietro al videogioco.

Anche per la parte audio troviamo diverse figure tra cui i compositori che si occupano di comporre il *tema musicale* principale del videogioco e i *sound designer* che hanno il compito di produrre e mixare i vari effetti e le musiche del gioco.

Come per gli ingegneri vi è una figura di leadership che assume il ruolo di direttore artistico.Il ruolo degli artisti è molto importante in quanto la qualità del loro lavoro influisce notevolmente sulla soddisfazione dell'utente finale.

Game designers

Il compito dei *game designers* è quello di concepire e progettare la parte interattiva dell'esperienza di gioco, nota anche come *gameplay*. Si possono individuare diversi compiti in basi ai diversi livelli di dettaglio. I game designer più esperti spesso si occupano di concepire la storia e l'evoluzione dei protagonisti nel corso del gioco. Si occupano di progettare la sequenza dei livelli e gli obiettivi da raggiungere per proseguire nel gioco. Altri game designer invece si occupano della progettazione specifica di alcune parti dei livelli e degli ambienti che compongono il videogioco. Alcuni designer lavorano a stretto contatto con gli ingegneri e i tecnici che si occupano del gameplay e a volte scrivono anche porzioni di codice (solitamente usando linguaggi di scripting ad alto livello). Spesso i game designer sono ex-progettisti che hanno scelto di ricoprire un ruolo più creativo nel concepimento di un videogioco.

2.2 Com'è strutturato un videogioco

Costruire un videogioco è un'attività che richiede prima un'importante fase di progettazione, perciò risulta importante capire com'è organizzato un videogioco. Qui di seguito è indicata una semplice architettura per un videogioco. Non tutti i videogiochi rispecchiano completamente quest'architettura, questo perché per questione di ottimizzazione e efficienza alcuni aspetti possono risultare un po' diversi, tuttavia tramite questo modello è possibile capire come funziona lo scheletro di un videogioco.

Partendo dall'alto possiamo dividere la struttura di un gioco in tre categorie principali:

- *Application layer*: uno strato che comunica direttamente con il sistema operativo e con l'hardware.
- *Logica di gioco*: si occupa di gestire lo stato del gioco e come questo cambia nel tempo.
- *Game view*: si occupa di presentare lo stato del gioco.

Questo tipo di architettura è molto simile al design pattern *Model View Controller* (*MVC*) che è basata sulla separazione tramite la parte computazionale, dell'interfaccia dalla parte dati di un'applicazione. Il modello

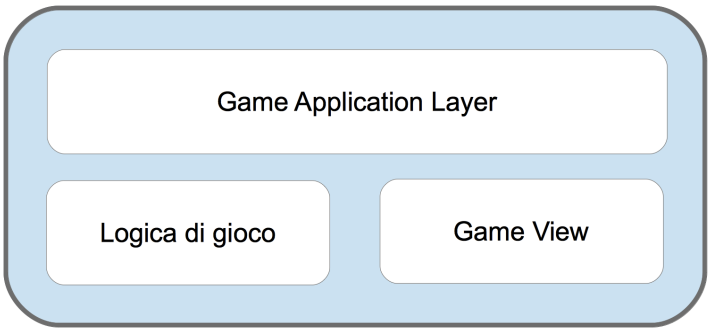

Figura 2.1: Architettura ad alto livello di un videogioco

indicato sopra ingloba il modello MVC e lo estende con uno strato che comunica con l'hardware e con il sistema operativo.

L'application layer è uno strato che isola la logica di gioco e la vista dal sistema operativo. Se perciò un giorno decidessimo di effettuare il porting del videogioco da un sistema operativo a un altro o da una console ad un'altra questo strato sarebbe in buona parte da riscrivere, ma gli altri strati non dovrebbero subire grosse variazioni. Nell'application layer troviamo il codice che comunica con i vari dispositivi hardware come ad esempio i controller usati per giocare, ma anche la gestione delle comunicazioni attraverso la rete o la gestione dei threads.

La logica di gioco è il gioco vero e proprio ed è separato dalla macchina su cui gira o da come è presentato al giocatore. In questo strato troviamo il sistema per gestire lo stato del gioco, per comunicare o rilevare cambiamenti di stato da altri sistemi. Sempre in quest'area possiamo trovare l'implementazione delle regole che compongono il mondo del nostro videogioco come ad esempio il motore fisico che permette di governare il modo in cui gli oggetti del nostro gioco si muovono o interagiscono.

La game view è responsabile della presentazione dello stato del gioco. Si occupa inoltre della traduzione degli input in comandi da inviare alla logica di gioco. A differenza dei livelli precedenti le game views possono essere molteplici. Possiamo ad esempio avere una view per i giocatori, per disegnare lo stato sullo schermo, per gestire gli input e per inviare i suoni agli altoparlanti. Ma possiamo avere anche una view per un giocatore remoto in rete o per i nemici dotati di intelligenza artificiale. Sebbene queste viste

facciano riferimento a diverse entità, fanno tutte capo alla stessa logica di gioco.

2.3 Application layer

L'application layer è lo strato più basso di un videogioco, è incaricato di diverse responsabilità tra cui: la lettura degli input, il caching delle risorse, la gestione della memoria, l'inizializzazione, la gestione e la conclusione del game loop. Approfondiamo ciascuna di queste parti.

2.3.1 Lettura degli input

I controller permettono al giocatore di interagire con il videogioco. È evidente che la pressione di un pulsante o comunque il cambio di stato di un controller possono avvenire in qualunque momento. Il problema di queste interruzioni asincrone come vedremo meglio successivamente viene affrontato tramite la lettura sincrona di una coda di eventi. Quando ad esempio il giocatore preme un pulsante questa azione viene inserita in una coda di eventi, questa coda viene svuotata e presa in carico dall'application layer ad ogni ciclo del game loop.

L'application layer ha anche il compito di tradurre lo stato delle periferiche di input in comandi. Questi comandi vengono poi inviati alla logica del gioco che provvederà ad eseguirli e determinerà in che modo questi si ripercuotono sullo stato del gioco.

Nella realizzazione di un gioco per computer o per console, è importante offrire al giocatore la possibilità di personalizzare le azioni associate ai pulsanti. Queste informazioni vengono salvate dall'application layer in un file di configuarazione e caricate ogni volta che viene avviato il gioco.

2.3.2 Caching delle risorse

Un videogioco fa un grande uso di risorse grafiche e sonore, come modelli 3D, texture, musiche, effetti audio. Affinché il videogioco scorra fluidamente è necessario che queste risorse siano caricate sulla memoria RAM del sistema, la quale risulta essere spesso limitata. Solitamente non è possibile caricare tutte queste risorse nella memoria del dispositivo, diventa allora importante una corretta gestione del caching delle risorse.

Analizzando il caso in cui il giocatore possa muoversi liberamente all'interno di un ambiente molto vasto (come ad esempio una città) è evidente come non sia possibile caricare in memoria tutte le risorse che compongono l'ambiente. Il sistema perciò carica una serie di risorse che ricoprano completamente la porzione dell'ambiente in cui si trova in quel momento il giocatore e mano a mano che questo si sposta il sistema deve cercare di capire in quale direzione si svolgerà il gioco e caricare le risorse che prevede siano necessarie a breve. Idealmente per il giocatore il gioco dovrebbe scorrere senza mai fermarsi a caricare.

Come abbiamo capito non basta progettare un sistema che sappia trovare le risorse e caricarle nel momento in cui siano necessarie. Il sistema deve essere intelligente e cercare di prevedere il futuro dello svolgimento del gioco, per fare ciò è necessario comunicare con la logica di gioco.

2.3.3 Gestione della memoria

La gestione della memoria è una problematica da non trascurare nella realizzazione di un videogioco. Quando si ha a che fare con la programmazione in C o in C++ è facile incappare in in problemi di memory leaks o memory corruption. Si parla di memory leaks ogni volta che viene allocata una porzione di memoria che non viene mai liberata, si verifica così uno spreco della memoria. Ogni qualvolta in cui il programma scrive dei dati su porzioni sbagliate di memoria abbiamo un caso di memory corruption, perdendo così i dati che vi erano stati memorizzati precendentemente. Una delle soluzioni più e usate per affrontare questa problematica è quella di estendere le funzioni basi di gestione della memoria offerte di base da sistema. Questa scelta, oltre ad ottenere buone prestazioni, permette di inserire delle features particolarmente utili in fase di debug. Ad esempio può tornare utile estendere la funzione di allocazione di memoria allocando una certa quantità di byte standard prima e dopo la porzione che conterrà i dati veri e propri. Scrivendo in queste due aree, in fase di allocazione, una sequenza standard di byte, successivamente in fase di debug sarà più facile individuare eventuali casi di memory corruption. Al momento della release, sarà opportuno modificare questa funzione, per limitare l'aggiunta di un solo byte prima e dopo la porzione di memoria.

2.3.4 Il game loop

Come dice il nome il game loop è un ciclo che si ripete per tutto il gioco. Possiamo semplificare la sequenza a tre operazioni principali:

1. Legge l'input queue, traduce gli input in comandi e li esegue nella sequenza in cui sono stati ricevuti.

2. Aggiorna lo stato del gioco tramite la logica di gioco. Questo step include l'elaborazione del physic engine, il collision detection e le elaborazioni d'intelligenza artificiale per determinare le azioni dei nemici.

3. Presenta lo stato attuale del gioco tramite le views. Questo significa effettuare il *rendering* della scena, suonare effetti sonori e eventualmente per i giochi online inviare sulla rete le modifiche avvenute allo stato del gioco.

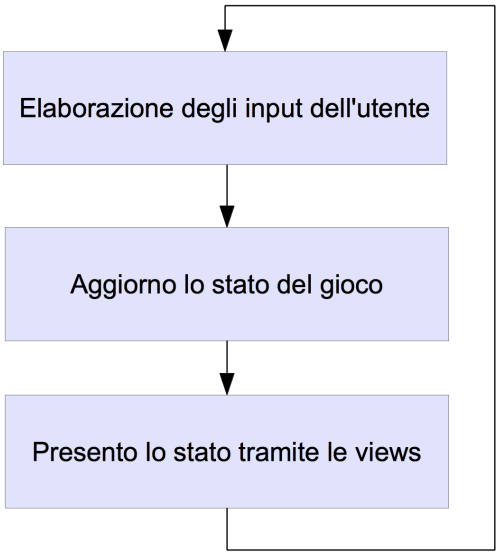

Figura 2.2: Struttura del game loop

Sui sistemi multicore alcune di queste operazioni possono essere assegnate a più *threads*. I threads possono essere usati per lo streaming audio, per

l'intelligenza artificiale e con alcune attenzioni persino per la fisica. È però necessaria un'opera di sincronizzazione al termine di ciascuna di queste 3 fasi. A volte è meglio limitare il numero di threads usati perché il tempo guadagnato con l'esecuzione in parallelo rischia di andare perso nell'attesa per terminare la sincronizzazione[11].

Quando il giocatore perde la partita o decide di terminare il gioco l'application layer si occupa di terminare il game loop.

2.4 La logica di gioco

La logica di gioco è il cuore del gioco e ne definisce l'universo: quali parti o entità lo compongano e come queste possano interagire tra di loro. In quest'area troviamo:

- le strutture che compongo lo stato del gioco e le procedure con cui è possibile modificarlo.
- la gestione della fisica e delle collisioni
- la gestione degli eventi
- l'interprete dei comandi

2.4.1 Stato del gioco e strutture dati

In tutti i giochi si può trovare un entità che contiene tutte le entità che compongono il gioco, cioè lo stato del gioco. Questi oggetti possono essere organizzati in diversi modi, tramite liste per i videogiochi più semplici. Quando però il progetto inizia ad ingrandirsi e complicarsi è meglio avere una struttura molto flessibile. Non bisogna trascurare che il nostro game engine poi dovrà scorrere rapidamente tra la struttura dati per leggere o modificare lo stato di un oggetto. Le soluzioni a disposizione sono molteplici a seconda delle esigenze, anche se solitamente non è possibile avere strutture facile da estendere in cui sia anche veloce trovare un determinato oggetto. Tra le soluzioni più utilizzate troviamo sicuramente le hash list e le strutture ad albero.

È importante non confondere l'entità e gli oggetti che compongono lo stato del gioco con la loro rappresentazione visuale. Come già discusso

precedentemente è bene tenere queste due parti separate. Un chiaro esempio viene dalla struttura dati di un personaggio di un giocatore che può contenere al suo interno informazioni come i punti vita o altri dati, queste informazioni compongono la struttura dati dell'entità e non hanno niente a che fare con la sua visualizzazione sullo schermo.

Le strutture dati per organizzare le diverse entità che compongo lo stato del gioco variano molto in base al tipo di gioco che si intende sviluppare.

2.4.2 Motore fisico e collision detection

Come le leggi della fisica governano il nostro universo se si intende realizzare un videogioco di alto livello e che rispecchi il mondo reale, è bene includere al suo interno un modello semplificato delle leggi della dinamica. Un motore fisico permette di inserire una serie di funzioni per simulare all'interno del gioco un *modello fisico newtoniano*. Grazie a queste funzioni una volta definito il comportamento del nostro universo (forze di gravità o altre forze che agiscono nel nostro ambiente) basterà definire per ciascuno oggetto le grandezze fisiche che lo rappresentano, come la massa, la velocità, l'accelerazione e la posizione e ciascun oggetto si muoverà e interagirà con l'ambiente seguendo le leggi indicate nel motore.

Tra le principali funzioni di un motore fisico troviamo anche il collision detection, che individua e segnala quando due o più oggetti collidono. Basta pensare alla dinamica di qualunque tipo di videogioco per capire quanto questa funzione venga utilizzata.

Un motore fisico molto preciso, offre un'esperienza di gioco molto realistica. Tuttavia oltre a richiedere un grande lavoro in fase di costruzione, richiede anche una notevole mole di calcoli quando viene mandato in esecuzione, questo può influire notevolmente sulle prestazioni del gioco. In fase di progettazione è importante valutare fino a che livello è opportuno affinare il motore fisico al fine di non appesantire troppo l'esecuzione del gioco col rischio che poi questo vada a scatti.

Visto la mole di lavoro che si nasconde dietro la costruzioni di un motore fisico esistono aziende che una volta costruito il loro decidono di metterlo in vendita per altri sviluppatori. Oltre alle soluzioni commerciali esistono anche diverse soluzioni open source a seconda delle esigenze. L'utilizzo di questi prodotti permette di guadagnare parecchio tempo nello sviluppo di un videogioco.

2.4.3 La gestione degli eventi

Ogni volta che la logica del gioco effettua delle modifiche sullo stato del gioco possiamo avere uno o più oggetti che reagiscono a tale azione. Ad esempio nella maggior parte dei casi in cui si verificano delle collisioni ci saranno sicuramente almeno uno o più oggetti che dovranno eseguire una determinata routine.

Una buona soluzione in questi casi è un'architettura basata ad eventi. I due principali punti di forza di quest'architettura sono: il disaccoppiamento tra la sorgente degli eventi e gli osservatori e la reattività che fa si che l'osservatore non debba chiedere in continuazione alla sorgente se è avvenuto l'evento, perché riceverà una notifica quando questo accade.

L'osservatore si registra presso la sorgente, manifestando così l'interesse nell'essere contattato quando si verifica un determinato evento. Quando avviene l'evento in questione la sorgente invierà tale notifica (tramite un messaggio) a tutti gli osservatori che si sono precedentemente registrati.

Questo tipo di architettura permette di tenere pulito il sistema che sta dietro al nostro gioco, risparmiando in molti casi il passaggio di riferimenti ad oggetti interessati ad un eventuale cambio di stato.

2.4.4 Interprete comandi

L'inserimento di un interprete comandi all'interno della logica di gioco può rivelarsi in alcuni casi una scelta intelligente. Un interprete comandi si occupa di tradurre un comando in un'azione concreta di uno o più oggetti.

Integrandolo con un'architettura ad eventi è possibile separare maggiormente la vista dalla parte logica di un gioco. Ad esempio in un gioco di automobilismo alla pressione di un pulsante al posto di associargli direttamente la chiamata di un metodo gli si associa l'invio di un comando. Questo comando viene catturato dagli oggetti interessati che assoceranno a questo una specifica azione. Standardizzando la gestione dei comandi è possibile sfruttarli anche con entità nemiche dotate di intelligenza artificiale.

Una volta implementato, questo sistema può essere esteso implementando una console in cui è possibile scrivere direttamente i comandi, questo sistema tornerà sicuramente utile in fase di testing e di debug.

Questa soluzione viene adottata molto nella programmazione di grossi giochi per computer.

2.5 Game views

Una game view è un sistema che comunicando con la logica di gioco, presenta lo stato del gioco ad un'osservatore. Per osservatore non si intende esclusivamente l'utente finale che sta giocando. Un osservatore può essere anche un agente con intelligenza artificiale che monitorando l'evoluzione dello stato del gioco determina il corso delle azioni di un nemico all'interno del videogioco. Ma possiamo avere una view anche per un giocatore in rete. Analizziamo ora questi tre casi.

2.5.1 Game view per l'utente

La game view che presenta lo stato del gioco al videogiocatore risponde agli eventi della logica di gioco, disegna la scena sullo schermo e invia gli effetti sonori e le musiche agli altoparlanti.

La grafica

La parte della view dedicata alla grafica si occupa di effettuare il *rendering* degli oggetti che compongono la scena di gioco. Il rendering è quel procedimento tramite cui è possibile generare un'immagine a partire da una serie di modelli 2D o 3D. Affinché il gioco scorra fluidamente il processo di rendering deve essere il più veloce possibile. I calcoli per il rendering confluiscono tutti sulla CPU e quando è disponibile vengono dirottati sulla GPU, ma questa non basta è importante scegliere a priori quali oggetti renderizzare e con che livello di definizione.

Prendiamo ad esempio il caso del rendering in un simulatore di volo. Quando l'aereo è fermo a terra, il sistema deve effettuare il rendering dei soli palazzi che lo circondano ma mano a mano che l'aereo prende quota la visuale si allarga e il numero di oggetti di cui fare il rendering cresce notevolmente. Ad un certo punto non è più possibile continuare a effettuare il rendering in tempi accettabili. L'unica soluzione per mantenere un buon numero di frame per secondo è quella di usare diversi livelli di dettagli per disegnare i diversi oggetti a seconda della loro distanza. Gli oggetti che si troveranno più vicino saranno renderizzati con un livello di dettagli elevato, mentre man mano che ci si allontana gli oggetti saranno renderizzati con una livello di dettagli via via inferiore.

Per non appesantire inutilmente la fase di rendering bisogna anche selezionare con cura quali elementi renderizzare e quali no. Prendiamo il caso di una porzione di città in cui è possibile esplorare anche gli interni delle case. Se il nostro giocatore si trova sulla strada nella fase di rendering non è necessario processare anche i modelli degli interni, sarebbe del calcolo inutile, lo stesso principio vale quando il nostro giocatore si trova all'interno o quando la visuale è oscurata da un muro. Non è necessario fare il rendering degli oggetti che sono nascosti alla vista dell'utente.

La progettazione e la scrittura di queste procedure per l'ottimizzazione del rendering è un lavoro molto complesso, ma in alternativa sono disponibili soluzioni commerciali.

Audio

La parte audio di un videogioco può essere divisa in tre principali categorie: effetti sonori, musiche e parlato.

Gli effetti sonori non sono particolarmente difficili da gestire. Spesso è sufficiente caricare il file e definire i parametri di riproduzione, come se l'esecuzione deve essere messa in loop. Inoltre la maggior parte delle consolle integrano oggi molteplici uscite audio per il collegamento ad un impianto surround, spesso offrono anche librerie in grado di riprodurre il suono al dallo speaker giusto in base alla posizione della sorgente sonora nella scena.

La gestione delle musiche non è particolarmente differente da quella degli effetti audio.

Per quanto riguarda la gestione dei dialoghi e del parlato è opportuno implementare un meccanismo di sincronizzazione con le immagini.

2.5.2 Game view per agenti AI

I sistemi ad agenti sono una delle soluzioni più utilizzate per l'organizzazione delle entità dotate di intelligenza artificiale all'interno del gioco come ad esempio i nemici. Questi devono reagire alle azioni del giocatore, per questo motivo è importante creare una vista anche per questi agenti.

Questa vista può variare notevolmente a seconda del tipo di gioco e a seconda del tipo di agente che si intende realizzare. In generale comunque si può definire uno scheletro abbastanza ricorrente. Prima di tutto occorre selezionare quali eventi si vuole monitorare, cioè a quali eventi si vuole

reagire. Tra questi oltre agli eventi che regolano le mosse e i movimenti del giocatore può essere utile anche monitorizzare gli eventi dell'ambiente circostante.

Una volta ricevuti gli eventi entra in gioco la parte decisionale dell'agente che individua come reagire agli stimoli ricevuti e quindi di conseguenza quali comandi inviare alla logica di gioco. Una semplice implementazione di questa parte potrebbe essere basata su una *macchina a stati*.

Gli agenti devono implementare funzioni per configurarli e apportargli modifiche. Deve essere ad esempio modificabile il livello di intelligenza e reattività così di poterlo aumentare via via che il gioco prosegue tra i livelli.

2.5.3 Game view per giocatori in rete

Per approfondire il concetto di vista per il gioco in rete analizziamo il caso di un giocatore che fa da host e di un giocatore remoto che si collega al suo computer o console. Da un lato avremo una macchina che fa da server

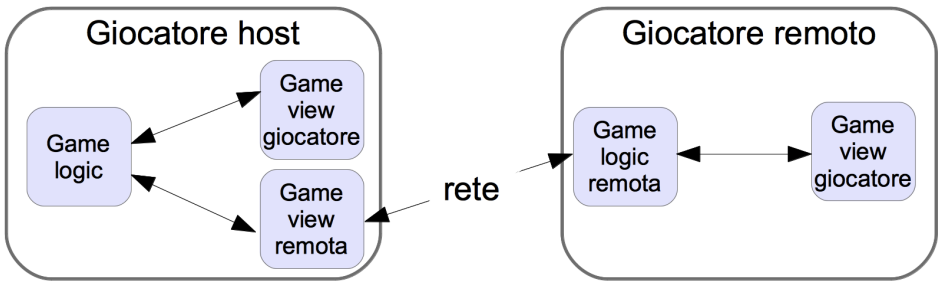

Figura 2.3: Gioco in rete tramite architettura client server

mentre dall'altro una macchina che si collega al server come client. Nonostante entrambe facciano girare lo stesso gioco, questo funziona con un regime differente.

Sulla macchina server il giocatore remoto appare come se fosse un giocatore gestito da un agente AI. Le informazioni e gli eventi della logica di gioco sono catturati e preparati dalla game view che seleziona queste informazioni evitando inutile ridondanza. Una volta raccolte le informazioni che riguardano l'attuale stato del gioco, vengono inviate sulla rete al giocatore remoto. Per inviare questi dati la game view interagisce con l'*application*

layer che spedisce al cient questi pacchetti sulla rete tramite il protocollo TCP o UDP. Se i pacchetti sono molto grandi, prima dell'invio vengono compressi.

La game view dedicata al giocatore remoto oltre a spedire le informazioni riguardo le variazioni dello stato del gioco deve ricevere anche i comandi che invia il giocatore remoto. Per questione di *sicurezza*, prima di mandarli in esecuzione, questi comandi devono essere verificati. Bisogna implementare un sistema di filtraggio per impedire che un utente remoto compia operazioni non ammesse.

Sulla macchina client la logica di gioco si comporta in modo molto simile a quella sul server, ma assume per buono lo stato del gioco della logica di gioco del server. Sul server infatti troviamo la logica del gioco finale. La logica di gioco sul client contiene comunque tutto il necessario per simulare il normale scorrimento del gioco. Rispetto alla sua opposta sul server, quella sul client ha anche una serie di funzioni per correggere il proprio stato del gioco, seguendo ciò che avviene sul lato server. Queste funzioni hanno che effettuano le correzioni hanno il permesso di superare le normali regole della logica di gioco. Questo è necessario, perché a causa degli immancabili ritardi che si verificano ogni qualvolta c'è di mezzo la rete, si possono verificare errori di consistenza tra i due stati di gioco. La logica di gioco remota interagirà poi con la game view remota inviandogli una serie di comandi e eventi proprio come succede dal lato server.

Capitolo 3

I game engines

Come si è visto nel capitolo precedente realizzare un videogioco richiede un grosso dispendio di risorse. I team che sviluppano applicazioni e videogiochi per smartphone sono spesso team composti da un numero limitato di persone. Per un team con risorse limitate sviluppare da zero un videogioco, spesso risulta essere troppo dispendioso e poco conveniente. In questi casi vengono in aiuto i game engines: framework che incorporano al loro interno buona parte dell'architettura comune ai videogiochi. L'utilizzo di questi framework semplifica e velocizza notevolmente lo sviluppo.

I game engine racchiudono già al loro interno l'application layer e buona parte della logica di gioco e delle views. All'interno di un game engine, l'application layer cioè lo strato più basso di un videogioco solitamente è già completamente implementato. Lo sviluppatore può realizzare il proprio videogioco avendo già a disposizione una serie di librerie con interfacce di alto livello. Ad esempio il meccanismo del game loop è già completamente implementato e solitamente i game engine tendono anche a nasconderlo. In questo caso lo sviluppatore dovrà limitarsi a scrivere la parte di codice per aggiornare lo stato del gioco cioè la logica di gioco. Anche nella scrittura di questa parte i game engine semplificano notevolmente il lavoro. infatti spesso al loro interno, è possibile trovare un motore fisico che si occupa di gestire tutta la parte di calcoli e il collision detection. I game engine spesso rendono disponibile una serie di meccanismi per la gestione degli eventi.

Quando si usano i game engine non sempre vi è una netta separazione tra la parte computazionale e la vista di un oggetto all'interno del gioco. Molti tendono a inglobare dentro un unico oggetto la sua parte visuale, cioè come

si manifesta poi al giocatore e anche la sua logica di gioco. Tra i pattern più ricorrenti possiamo individuare sicuramente il pattern Composite. Grazie a questo pattern si riesce a separare per ciascun oggetto la sua vista dalle sua parte logica.

Sul mercato sono disponibili diversi game engines per diverse piattaforme. Per gli smartphones. vi sono diverse soluzioni commerciali, ma non mancano anche game engines gratuiti rilasciati sotto licenza open source. I game engines più interessanti e più utilizzati sono: Unity 3D, Corona, Sparrow e Cocos2D. Analizziamo ciascuno di questi tramite le informazioni reperibili sui rispettivi siti web [17] [15] [1] [13].

3.1 Unity3D

Unity3D è uno dei game engine più utilizzati per realizzare videogiochi non solo per iPhone. Unity3D permette infatti di esportare il proprio progetto per tantissime piattaforme tra cui: Mac OSX, Windows, iOS, Android, PlayStation 3, XBox 360, Wii e anche il web. Come dice il nome, questo game engine è nato per creare videogiochi in 3D, ma con alcuni semplici stratagemmi è possibile realizzare anche giochi 2D. Unity è un software proprietario ed è disponibile in diverse versioni tra cui una gratuita ma limitata.

Unity3D integra un editor *WYSIWYG* (what you see is what you get) attraverso il quale lo sviluppatore può comporre la scena o l'ambiente trascinando e manipolando direttamente gli oggetti.

Il rendering è basato sul recente paradigma *deferred rendering*. Questo metodo si basa sul disaccoppiamento del rendering della scena dal calcolo delle luci. Nel deferred rendering vengono renderizzati tutti gli oggetti visibili ignorando l'illuminazione che li colpisce, il risultato di questo primo passaggio viene memorizzato in un buffer. Successivamente avviene il calcolo dell'illuminazione che grazie al lavoro svolto precedentemente avviene solo per gli oggetti o le porzioni di oggetti visibili. Grazie alla separazione delle due fasi è possibile evitare calcoli inutili per l'illuminazione (una delle fasi più pesanti).

Unity3D incorpora al suo interno il motore fisico *PhysX* sviluppato da NVIDIA, uno dei motori fisici più avanzati. Oltre alla normale gestione

della fisica dei corpi rigidi, permette anche di simulare il movimento dei vestiti dei personaggi animati e dei soft bodies.

Nelle ultime versioni integra potenti strumenti per il *pathfinding*. Queste features sono estramamente utili nella gestione dell'intelligenza artificiale. Con poco lavoro sarà possibile far muovere una mesh (cioè un modello 3D) nello spazio automaticamente, evitando ostacoli e seguendo percorsi dinamici.

Lo sviluppatore può scegliere di usare tre diversi linguaggi di programmazione: *C#*, *JavaScript* e *Boo* (un linguaggio di programmazione ispirato a Python). Tutti questi tre linguaggi possono fare uso delle *librerie .NET*. I linguaggi di scripting sono spesso accusati di essere lenti, per ovviare a questo problema Unity compila tutto in codice nativo che riesce ad eseguire molto velocemente. Lo scambio di messaggi tra i vari moduli che compongono il videogioco avviene tramite un sistema ad eventi.

Vale la pena segnalare l'*asset store*, un negozio online, che permette di acquistare e vendere risorse per i videogiochi, tra queste possiamo trovare modelli 3D, textures, suoni ma anche script già pronti. Il tutto è pienamente integrato con il software di sviluppo.

Sul sito ufficiale è possibile trovare una documentazione molto approfondita e una serie di tutorial per creare un proprio videogioco da zero. Unity3D vanta una curva di apprendimento molto ripida, cioè in poche ore di studio e pratica è possibile realizzare già dei videogiochi base.

Come scritto precedentemente Unity3D è un software proprietario, sono disponibili diverse licenze a seconda delle esigenze. La versione gratuita permette di rilasciare i giochi per Windows, Mac e anche per i browser che installano un apposito plugin. La licenza base per esportare i propri progetti per iOS costa 280 € e altrettanto per la piattaforma Android. È a disposizione anche una versione pro che include diverse funzionalità non disponibili nella versione base tra cui: il pathfinding, funzioni avanzate di rendering e la possibilità di alleggerire in termini di dimensioni il file eseguibile eliminando parti inutilizzate del game engine. Unity3D pro costa 1050 € a cui vanno aggiunti altri 1000 € per la licenza pro di iOS o di android Android.

Per i piccoli sviluppatori la spesa iniziale può risultare eccessiva. Inoltre Unity3D non integra un sistema per l'animazione degli sprite 2D, ma sono disponibili diversi script che si occupano di questa funzione.

In conclusione Unity3D è uno strumento potentissimo per realizzare gio-

chi in tre dimensioni. Se però si intende sviluppare giochi in due dimensioni l'offerta di strumenti e la spesa può risultare eccessiva e vale la pena analizzare anche altre soluzioni.

3.2 Shiva3D

Shiva3D è un game engine per realizzare videogiochi in tre dimensioni. Shiva3D permetter di esportare il proprio progetto per molteplici piattaforme tra cui: Mac OSX, Windows, iOS, Android, Palm WebOS, Wii e il web. Anche Shiva è un software proprietario ed è disponibile in diverse versioni tra cui una gratuita ma limitata alla sola pubblicazione per il web.

Il linguaggio principale per la realizzazione dei progetti in Shiva3D è una versione ottimizzata di *Lua*. Lua è uno scripting language multi paradigma con semantica estendibile. Per via della sua leggerezza e della sua velocità, fa si che sia molto utilizzato nell'ambito dei videogiochi. Shiva comunque lascia la possibilità agli sviluppatori di usare anche altri linguaggi come C, C++ e Objective-C.

Anche Shiva3D mette a disposizione degli sviluppatori un editor WYSIWYG per costruire rapidamente le scene e gli ambienti che compongono il videogioco.

Per quanto riguarda la fisica, Shiva3D si affida al motore grafico *ODE*. ODE (Open Dynamic Engine) è un motore fisico open source platform indipendent. Permette complesse simulazioni per corpi rigidi e integra anche funzioni per rilavare le collisioni. Purtroppo questo motore fisico non permette nessuna simulazione dei cosiddetti soft bodies, cioè corpi semirigidi, come ad esempio i tessuti di un vestito.

Il rendering è affidato ad un motore proprietario sviluppato internamente basato su una *forward rendering* pipeline.

Sul sito ufficiale è disponibile un'approfondita documentazione. Sono presenti anche un notevole numero di tutorial per prendere confidenza con il linguaggio Lua. Infine sempre sul sito ufficiale sono disponibili guide apposite per gli sviluppatori che provengono dall'engine Unity3D.

La licenza free permette di pubblicare i progetti esclusivamente per il web. La licenza Basic in vendita per 400 $ permette di pubblicare il proprio videogioco su tutte le piattaforme supportate da Shiva3D, cioè: Mac OSX, Windows, iOS, Android, Palm WebOS e Wii. È presente anche una licenza

Advance che al prezzo di 2000 $ rispetto alla versione basic offre anche il supporto al sistema *SVN* e opzioni di ottimizzazioni dei dettagli nella fase di rendering. Tutte queste licenze non prevedono successive royalties.

Anche Shiva3D non integra nativamente funzioni per animare sprite 2D, ma nella community degli sviluppatori è possibile trovare script già pronti.

3.3 Corona

Corona è un game engine per lo sviluppo di giochi in due dimensioni. Corona è nato con la missione di unificare lo sviluppo di applicazioni per le piattaforme iOS e Android.

A differenza dei game engine visti precedentemente, Corona non dispone di un editor WYSIWYG in cui sia possibile trasinare gli oggetti all'interno della scena, tuttavia, man mano che si scrive il codice è possibile visualizzare in un simulatore come cambia la composizione della scena.

Anche in questo game engine il linguaggio di riferimento è *Lua*. Corona mette a disposizione una serie di librerie con interfacce di alto livello, permettendo agli sviluppatori di realizzare un piccolo videogioco in pochissimo tempo.

Come motore fisico Corona si affida a *Box2D*. Box2D è un physic engine open source per simulazioni di fisica in due dimensioni. Questo motore fisico scritto in C++ è platform indipendent. Include al suo interno il rilevamento delle collisioni. Può gestire solo la fisica dei corpi rigidi.

Corona è un software proprietario ed è disponibile con tre divese licenze. La versione pro, al costo di 349 $ all'anno, permette esportare i propri videogiochi per le piattaforme iOS, Android ma anche per il tablet Kindle Fire di Amazon e per l'ebook reader nook prodotto da Barnes & Noble. In alternativa sono disposnibile anche le singole licenze per la piattaforma iOS o Android al costo di 199 $ all'anno per ciascuna.

Inclusa in tutte le licenze vi è anche l'iscrizione al sistema di promozione LaunchPad. LaunchPad è un portale sviluppato da Corona per promuovere le proprie applicazioni, inoltre permette agli sviluppatori di analizzare l'uso che ne fanno gli utenti del videogioco sviluppato.

Corona grazie al linguaggio Lua vanta una ripida curva d'apprendimento. Inoltre permette di scrivere un videogioco una volta ed esportarlo sulle principali piattaforme per smartphones al giorno d'oggi.

Tuttavia questo sistema viene anche molto criticato. L'impossibilità di usare qualunque linguaggio in alternativa a Lua, viene vista come una grossa limitazione. Le librerie messe a disposizione da Corona sono di alto livello e spesso, gli sviluppatori più esperti, lamentano l'impossibilità di bypassare queste api che alle volte, tendono a nascondere troppo in fase di sviluppo.

3.4 Cocos2D

Cocos2D è un framework *open source* per lo sviluppo di videogiochi in due dimensioni. Il progetto origale era scritto in Python, ma si sono sviluppati diversi porting ad altri linguaggi e piattaforme. Tra i vari progetti derivati, quello più evoluto e più solido è sicuramente *Cocos2D-iPhone*.

Cocos2D-iPhone è un porting in linguaggio *Objective-C* per iOS (iPhone, iPad e iPod Touch) e Mac OS X. Segue gli stessi concetti e lo stesso design del framework originale. Nell'ultima versione fa uso della librerie OpenGL ES 1.1. Questo framework si pone proprio sopra queste librerie e permette agli sviluppatori di realizzare i propri giochi, "toccarle" mai direttamente. Tuttavia essendo open source si può analizzare ed eventualmente modificare o espandere, il codice che si nasconde dietro al motore.

Cocos2D supporta due motori fisici: *Box2D* e *Chipmunk*. Entrambi i motori supportano la fisica dei corpi rigidi e sono entrambi open source. Box2D è scritto in C++ mentre Chipmunk è scritto in C. Gli sviluppatori sono liberi di scegliere se e qual incorporare nei propri giochi.

A differenza di altri engine, Cocos2d non dispone di un editor WYSIWYG e lo sviluppatore rispetto ad altri engine, ha sicuramente più a che fare con la scrittura di codice in Objective-C. Tuttavia sono disponibili una serie di tool (alcuni gratuiti e altri a pagamento) per velocizzare la creazione di menu o di scene manipolando direttamente gli oggetti che li compongono.

Come già scritto il progetto è open source e la community è invitata a contribuire allo sviluppo del game engine. Il fondatore del progetto per la piattaforma iOS è Ricardo Quesada. Ricardo insieme a Rolando Abarca, è tra i maggiori contributori. A partire da maggio 2011 i due sviluppatori sono stati assunti da *Zynga*, ma continuano a contribuire al progetto lavorando alla versione 2.0 [10]. Zynga è una software house molto conosciuta per il successo riscosso con i videogiochi lanciati sul social network facebook.

Nell'ultimo anno ha rilasciato anche le versioni iPhone dei suoi titoli di maggior successo, basando il proprio lavoro sul framework Cocos2D.

A differenza di molti altri progetti open source nati sulla rete, la documentazione fornita dal sito ufficiale del progetto è abbastanza completa. Inoltre Cocos2D-iPhone ha una community di utenti molto vasta e molto attiva sul forum ufficiale e su altri portali. Sono stati pubblicati anche molti libri per realizzare videogiochi utilizzando questo framework.

Vale la pena segnalare anche il progetto Cocos2D-Android, con lo scopo di effettuare il porting delle librerie al linguaggio Java per la piattaforma Android, tuttavia allo stato attuale questo framework è ancora molto acerbo e poco affidabile.

Cosos2D è uno tra i game engine più utilizzati per sviluppare videogiochi in due dimensioni per la piattaforma iOS [10]. A suo favore giocano sicuramente il fatto di essere gratuito e la facile estendibilità grazie ai sorgenti aperti.

Capitolo 4

Caso di studio su iPhone

In questo capitolo si analizza lo sviluppo di un piccolo videogioco per iPhone. Partendo dall'idea, si analizzerà la struttura e gli elementi che compongono il gioco.

Figura 4.1: Uno screenshot del gioco finale

L'idea per il progetto era quella di realizzare un piccolo videogioco che

facesse uso dei *sensori* installati su un iPhone. Si è perciò deciso di realizzare un gioco in cui il giocatore comanda un'astronave che deve schivare degli asteroidi oppure spararegli per farli esplodere.

Il giocatore interagisce con il gioco tramite l'*accelerometro* e il *touch screen*. Tenendo in mano direttamente il telefono, l'utente può inclinare il dispositivo per far muovere l'astronave nella direzione da lui desiderata. Toccando lo schermo in qualunque punto l'astronave può sparare agli asteroidi.

Il gioco deve quindi gestire diversi tipi di *input*.

La dinamica del gioco prevede quindi che l'astronave viaggi in direzione rettilinea a velocità costante e che incontri degli asteroidi lungo il percorso. L'utente inclinando il telefono potrà muovere l'astronave all'interno dello schermo per schivare gli asteroidi. Il giocatore, toccando lo schermo può anche far sparare l'astronave. Quando lo schermo verrà toccato in punto qualunque la navicella sparerà un proiettile dalla posizione in cui si trova, il proiettile viaggerà in direzione rettilinea e se per caso questo si scontra con un asteroide, questo dovrà esplodere e il proiettile dovrà sparire. Per ogni asteroide colpito il giocatore guadagnerà un certo numero di punti. Ogni volta che l'astronave sarà colpita da un'asteroide il giocatore perderà una vita. Quando le vite del giocatore sono terminate il gioco si ferma e al giocatore viene mostrato un messaggio di game over con il punteggio totalizzato. Il giocatore potrà quindi decidere di giocare nuovamente o tornare al menù iniziale.

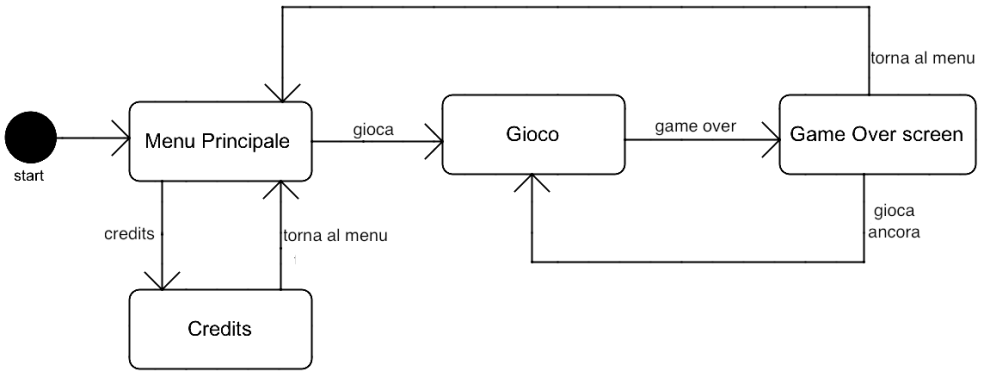

Figura 4.2: Diagramma di stato delle schermate dell'applicazione

L'applicazione sarà corredata da una serie di menù che guideranno l'utente. Nel diagramma mostrato in figura 4.2, vediamo come le diverse schermate si possono alternare durante l'esecuzione dell'applicazione. Tra questi oltre al gioco vero e proprio, troviamo il menu principale, una schermata con i credits del videogioco e una schermata di game over.

Nel diagramma non compare nessun nodo che indica la fine dell'applicazione, questo perché non c'è una vera fine nel flusso di esecuzione dell'applicazione. L'utente può infatti premere in qualunque momento il *tasto home* dell'iPhone con cui l'applicazione viene chiusa. Tutto questo è gestito direttamente dal sistema operativo.

4.1 Gli elementi che compongono il gioco

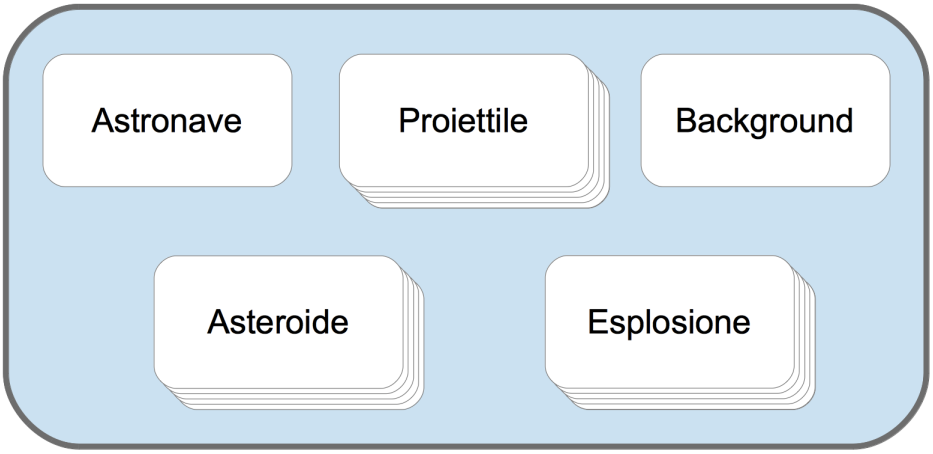

Figura 4.3: Gli oggetti che compongono il videogioco

Possiamo individuare fin da subito una serie di oggetti che compongono la scena principale del videogioco. Sicuramente all'interno del gioco avremo un'astronave, dei proiettili, degli asteroidi e delle esplosioni infine dietro a tutti questi oggetti vogliamo che ci sia un'immagine di sfondo. Tutti questi oggetti hanno una rappresentazione grafica sullo schermo, ma come vedremo successivamente sarà necessario introdurre altri oggetti, privi di rappresentazione, per organizzare meglio il gioco.

L'*astronave* è l'entità con cui il giocatore si affaccia sul gioco. Il giocatore può infatti muovere l'astronave e farla sparare. L'entità astronave dovrà ovviamente avere una rappresentazione grafica e dovrà rispondere al segnale dell'accelerometro e agli eventi del touch screen. Tuttavia come abbiamo visto nel secondo capitolo, la gestione degli input affidata direttamente all'entità che rappresenta il giocatore non è la scelta migliore. L'astronave dovrà limitarsi a rispondere a una serie di *comandi*. Occorre quindi introdurre nella scena principale un'entità, priva di visualizzazione che abbia il compito di codificare gli input in comandi.

I *proiettili* sono da considerare entità separate rispetto all'astronave. Nonostante questi siano sparati da essa, hanno vita separata. I proiettili infatti dopo il lancio non rispondono in alcun modo ai comandi dell'utente e la loro traiettoria è totalmente indipendente dall'astronave.

Il giocatore può sparare a raffica una serie di proiettili quindi in uno stesso momento sulla scena del gioco possono essere presenti più proiettili, ognuna di questi è una singola entità che non ha nulla a che fare con gli altri proiettili. Tuttavia può essere utile una struttura che possa organizzare questi proiettili, allocandone inizialmente una certa quantità.

Gli *asteroidi* sono entità con una rappresentazione grafica. Come per i proiettili, possiamo avere più istanze di questa entità presenti contemporaneamente sulla scena. Quindi anche in questo caso può essere utile introdurre un'entità che organizzi e gestisca tutti gli asteroidi. Questa nuova entità avrà il compito di allocare gli asteroidi, lanciare gli asteroidi quando richiesto e controllare se gli asteroidi entrano in collisione con l'astronave o con i proiettili. Per effettuare questo controllo, dovrà quindi accedere alla posizione dell'astronave e di tutti i proiettili attivi in quel momento. Quando un'asteroide entra in collisione con un proiettile o con l'astronave deve esplodere. Ciascun asteroide deve quindi poter mostrare anche la rappresentazione di un'*esplosione*.

Dietro a tutti questi oggetti vi sarà uno sfondo. Lo sfondo sarà costituito da un'immagine. Deve essere possibile modificare l'immagine dello sfondo sostituendola con una nuova. Il *background* deve poter scorrere, dando così l'idea che l'astronave si stia muovendo.

All'interno del gioco vi deve essere infine un entità, priva di rappresentazione grafica, che si occupi di azionare gli asteroidi. Questa entità potrà quindi decidere come, quando e dove lanciare un asteroide. Il sistema che prenderà queste decisioni potrà essere più o meno complicato a seconda del

livello di difficoltà che si vuole implementare all'interno del gioco. Questo sistema deve in qualche modo accedere agli asteroidi per poterli lanciare e volendo anche ai movimenti del giocatore per poterlo mettere in difficoltà.

Tutti questi oggetti fanno parte della scena principale del gioco. Questa scena dovrà anche gestire lo stato di game over del gioco e la possibilità di cominciare una nuova partita.

4.2 Scelta del game engine

Per velocizzare lo sviluppo del gioco si è deciso di usare un game engine. Il game engine scelto per il progetto è *Cocos2D*. Essendo il gioco da realizzare in due dimensioni sono stati presi in considerazione principalmente i framework: Corona e Cocos2D. Gli altri game engine analizzati nei capitoli precedenti come Unity3D e Shiva3D sono stati scartati. Questi due, sono nati per sviluppare giochi in tre dimensioni e sebbene si prestino anche alla realizzazione di giochi in 2D, i costi delle licenze sono sicuramente eccessivi. Anche Corona, il framework sviluppato da Ansca è stato scartato per via della sua licenza commerciale e delle interfacce di livello un troppo alto.

La versione 1.0.1 del porting del framework Cocos2D ai dispositivi iOS si è rivelata la scelta migliore. Questo game engine è molto maturo ed essendo *open source* è stato possibile analizzare anche la struttura che vi sta dietro.

4.2.1 Cocos2D in dettaglio

Analizziamo il funzionamento del game engine Cocos2D partendo dalla *scene graph*. Per scene graph si intende quella struttura che permette di organizzare gli oggetti in una scena. Cocos2D adotta una *struttura ad albero* dove i nodi sono oggetti derivati dalla classe *CCNode*. Molti nodi hanno una rappresentazione grafica che verrà poi raffigurata sullo schermo, tuttavia esistono anche nodi privi di una rappresentazione, come ad esempio CCScene e CCLayer.

I nodi che compongono la scena sono organizzati con un meccanismo padre figlio. Ciascun nodo può avere più figli ma ha al massimo un padre. Questa relazione padre figlio non ha niente a che fare con l'ereditarietà delle classi nei linguaggi object oriented.

Questa gerarchia ad albero viene detta scene graph. È evidente come Cocos2D faccia uso del pattern *composite* per organizzare i nodi.

La classe CCNode

CCNode è la classe base per tutti i nodi. In figura 4.4 sono presenti alcune

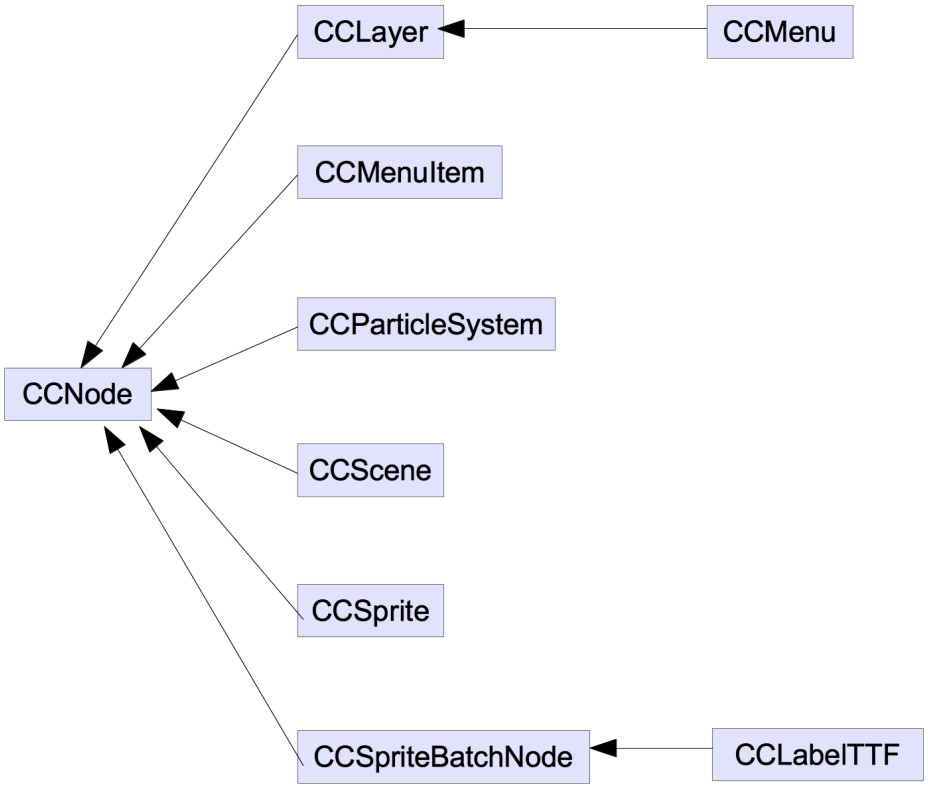

Figura 4.4: Alcune delle molte classi che derivano da CCNode

delle tante classi che derivano da CCNode.

CCNode è una classe priva di una concreta rappresentazione sullo schermo. Definisce e implementa tutti i metodi per aggiungere, recuperare e rimuovere nodi figli. Ecco alcuni dei modi e dei metodi con cui si può interagire con i nodi:

- Per creare un nuovo nodo:
  CCNode* myNode = [CCNode node];

- Per aggiungere un nodo come figlio:
  [mynode addChild:childNode z:0 tag:123];
- Per recuperare un nodo figlio:
  CCNode* retrievedNode = [myNode getChildByTag:123];
- Per rimuovere un nodo figlio tramite il suo tag:
  [myNode removeChildByTag:123 cleanup:YES];
- Per rimuovere tutti i figli di un nodo:
  [myNode removeAllChildrenWithCleanup:YES];
- Per rimuovere un nodo dal proprio nodo padre:
  [myNode removeFromParentAndCleanup:YES];

il parametro z nel metodo addChild, detto anche *z order*, determina l'ordine in cui i nodi vengono disegnati. Il nodo col valore z più basso sarà disegnato per primo, quello col valore più alto sarà disegnato per ultimo. Se più nodi hanno lo stesso valore di z, questi saranno disegnati nell'ordine in cui sono stati aggiunti al nodo padre. Tutto questo ovviamente si applica ai soli nodi che hanno una rappresentazione grafica.

Il parametro *tag* invece permette di identificare e poi riottenere uno specifico nodo. Se più nodi sono inseriti con lo stesso tag, il metodo getChildByTag restituirà sempre e solo il primo nodo. I rimanenti nodi diventeranno così inaccessibili. È quindi importante usare tag unici per identificare i nodi.

Lo scheduling dei messaggi

Sui nodi si possono programmare la ricezione di messaggi. In Objective-C per invio di un messaggio si intende la chiamata di un metodo di un oggetto. Capita spesso che si richieda la chiamata pianificata ad un particolare metodo di un nodo, questo permette di pianificare una serie di operazioni, come ad esempio la rilevazione di collisioni o il movimento di un'immagine. Il più delle volte si programma la chiamata del metodo update ad ogni frame, per fare ciò basta chiamare il metodo scheduleUpdate sul nodo desiderato o sul nodo stesso in cui si esegue la chiamata:

```
[self scheduleUpdate];
```

Il nodo in questione dovrà quindi implementare il metodo:

```
-(void) update:(ccTime) delta {
       // questo metodo verrà eseguito ad ogni frame
}
```

Il parametro delta indica il tempo trascorso dall'ultima volta che il metodo è stato chiamato. In alternativa si può pianificare anche la chaimata di altri metodi ad altri intervalli, ad esempio per programmare la chiamata di un metodo ogni 2 secondi basterà chiamare sul nodo il metodo:

```
[self schedule:@selector(updateOgniDueSecondi:) interval:2.0f];
```

e implementare nel nodo il metodo indicato nella precedentemente chiamata:

```
-(void) updateOgniDueSecondi:(ccTime)delta {
       // questo metodo verrà chiamato ogni 2 secondi
}
```

Se l'intervallo viene settato a 0, il metodo verrà chiamato ad ogni frame, come avviene per il metodo update.

Per fermare la chiamata programmata del metodo update basterà la chiamata del metodo:

```
[self uncheduleUpdate];
```

È possibile anche terminare la chiamata programmata di un determinato metodo tramite:

```
[self unchedule:@selector(updareOgniDueSecondi:)];
```

Infine per terminare la chiamata programmata di tutti i metodi il metodo implementato da tutti i nodi è:

```
[self unscheduleAllSelector];
```

È possibile anche assegnare una *priorità* ai metodi update.

```
// nel nodo A
-(void) scheduleUpdates {
       [self scheduleUpdate]; // di default priority è settato a 0
}

// nel nodo B
-(void) scheduleUpdates {
```

```
        [self scheduleUpdateWithPriority:-1];
}

// nel nodo C
-(void) scheduleUpdates {
        [self scheduleUpdateWithPriority:1];
}
```

Nella porzione di codice qui sopra, i 3 nodi pianificano la chiamata programmata del metodo update. Questo verrà eseguito prima sul nodo B, poi sul nodo A e successivamente sul nodo C. I metodi update sono chiamati procedendo da quello col valore di priorità inferiore a quello maggiore. Quando non è specificata alcuna priorità, la chiamata dei metodi update segue lo z order, cioè l'ordine in cui i nodi sono disegnati. L'uso delle priorità nelle chiamate dei metodi update è una cattiva pratica di programmazione e viene usata raramente, in quanto è comunque difficile da gestire e può indurre errori difficili da individuare.

Director, scene e layers

La classe *CCDirector* implementa il *Director*: il cuore del game engine. Questa classe è un *singleton*, questo significa che nello stesso momento non può esistere più di un'istanza della classe CCDirector, questa istanza è accessibile globalmente tramite il metodo sharedDirector.

Il CCDirector mantiene in memoria le configurazioni del game engine e permette di gestire le scene. Attraverso le varie operazioni che il director mette a disposizione, possiamo:

- Accedere alla scena in esecuzione
- Mandare in esecuzione o sostituire una scena
- Accedere alle configurazioni di Cocos2D
- Mettere in pausa, riprendere o concludere l'esecuzione del gioco

Nella gerarchia dei nodi del scene graph il primo nodo è sempre una *CCScene*. Questa classe racchiude il concetto astratto di scena. La classe CCScene virtualmente non contiene nessuna funzione addizionale rispetto a CCNode. Tuttavia la classe CCDirector richiede un'oggetto istanziato dalla

classe CCScene (o da una classe derivata da questa), per settare la nostra scena come l'attuale scena attiva visualizzata sullo schermo.

Normalmente i figli di un nodo CCScene derivano dalla classe CCLayer, che normalmente contiene gli oggetti che compongono il gioco. La scena nella maggior parte dei casi non contiene direttamente porzioni di codice del gioco, perciò è prassi creare un metodo statico: +(id) scene dentro ad un'oggetto CCLayer che restituisca un oggetto CCScene che ha già come figli i vari layer, ad esempio:

```
+(id) scene {
        CCScene *scene = [CCScene node];
        CCLayer* layer = [HelloWorld node];
        [scene addChild:layer];
        return scene;
}
```

Per far sì che la nostra scena venga poi visualizzata all'avvio della applicazione dovremo inserire nella classe *AppDelegate*, nel metodo applicationDidFinishLaunching il codice:

```
[[CCDirector sharedDirector] runWithScene:[HelloWorld scene]];
```

Quando modifichiamo questo metodo nella classe AppDelegate è come se agissimo sul metodo *main* di un programma scritto in Java o in C.

Successivamente per rimpiazzare la scena precedente con una nuova scena useremo il metodo:

```
[[CCDirector sharedDirector] replaceScene: myNewScene];
```

Quando si carica una nuova scena, questa viene prima caricata completamente nella memoria del dispositivo, solo al termine di questo caricamento viene liberata la memoria occupata dalla scena precendente. Una transizione da una scena all'altra, per un piccolo lasso di tempo occupa una porzione di memoria grande come le due scene messe assieme. Questa situazione può generare dei memory warnings da parte del sistema operativo alla nostra applicazione. Successivamente analizzeremo una soluzione per aggirare questo problema.

I *CCLayer* sono utili per organizzare gli oggetti sulla scena. Ad esempio si può creare un layer per contenere lo sfondo e un altro layer contenente tutti gli oggetti del gioco. Il layer con lo sfondo viene aggiunto alla scena

e successivamente anche quello con gli oggetti. In questo modo tutti gli oggetti saranno disegnati dopo il backgorund, quindi sopra di esso.

Come già visto precedentemente per organizzare gli oggetti non è necessario un oggetto derivato dalla classe CCLayer, ma spesso può bastare un semplice CCNode. Questa soluzione è spesso preferibile quando c'è solo bisogno di organizzare gli oggetti.

Ricezione degli eventi di input

Uno dei maggiori vantaggi della classe CCLayer è che permette di ricevere gli *input* del touch screen. Per abilitare la ricezione degli eventi basta settare a YES la proprietà isTouchEnabled:

```
self.isTouchEnabled = YES;
```

Questa abilitazione può essere settata in qualunque momento, ma solitamente si setta all'interno del metodo init. Una volta attivata la ricezione degli input del touch screen, il layer può implementare una serie di metodi per gestire i diversi eventi:

- Quando un dito inizia a toccare lo schermo viene chiamato il metodo:
  -(void) ccTouchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
- Quando il dito si muove sullo schermo viene chiamato il metodo:
  -(void) ccTouchesMoved:(NSSet *)touches withEvent:(UIEvent *)event
- Quando il dito lascia lo schermo viene chiamato il metodo:
  -(void) ccTouchesEnded:(NSSet *)touches withEvent:(UIEvent *)event

Quando lo schermo viene toccato spesso si vuole conoscere il punto in cui lo schermo viene toccato. Gli eventi sono ricevuti dalle API del framework Cocoa Touch, successivamente le cordinate del punto devono essere convertite in coordinate OpenGL, ad esempio col seguente metodo:

```
-(CGPoint) locationFromTouches:(NSSet *)touches {
       UITouch *touch = [touches anyObject];
       CGPoint touchLocation = [touch locationInView: [touch view
              ]];
       return [[CCDirector sharedDirector] convertToGL:
              touchLocation];
}
```

Gli oggetti istanziati dalla classe CCLayer, se abilitati, possono ricevere anche gli eventi dell'accelerometro:

```
self.isAccelerometerEnabled = YES;
```

Il layer per gestire gli eventi dell'accelerometro deve poi implementare il metodo:

```
-(void) accelerometer:(UIAccelerometer *)accelerometer
       didAccelerate:(UIAcceleration *)acceleration
{
       NSLog(@"Accelerazione x:%f / y:%f / z:%f", acceleration.x,
              acceleration.y, acceleration.z);
}
```

Accedere alle tre componenti dell'accelerazione è molto semplice, il codice qui sopra è da esempio.

Gli sprite

Una delle classi più usata nella realizzazione di un videogioco è la classe *CCSprite*. Questa classe permette di visualizzare sullo schermo un'immagine:

```
CCSprite* sprite = [CCSprite spriteWithFile:@"file.png"];
[layer addChild:sprite];
```

La posizione dello sprite di default sarà 0,0, cioè sarà posizionato nell'angolo in basso a destra dello schermo. Per modificare successivamente la posizione dello sprite basta modificare l'attributo position.

La gestione della memoria in Objective-C e in Cocos2D

In Objective-C la memoria è gestita tramite *reference counting*. Per allocare un'oggetto basta chiamare metodi di classe alloc:

```
MyClass *myObject = [[MyClass alloc] init];
```

Nell'esempio qui sopra l'oggetto myObject viene allocato e inizializzato. Per liberare la memoria occupata dall'oggetto è disponibile il metodo dealloc.

Questo metodo però non viene mai chiamato esplicitamente, ma verrà chiamato indirettamente dal metodo release. Ogni oggetto ha infatti un contatore detto *retain count*. Il retain count indica se all'interno del programma c'è ancora qualcuno interessato ad usare il nostro oggetto.

Quando il nostro oggetto viene allocato il retain count viene inizializzato a 1, gli altri oggetti interessati ad usare il nostro oggetto eseguiranno in metodo retain su di esso per poterlo trattenere, questo incrementa il retain counter di 1. Successivamente quando non si ha più bisogno dell'oggetto in questione, questo deve essere rilasciato, chiamando su di esso il metodo release, che decrementerà il contatore di 1. Quando il contatore arriverà a 0 il metodo dealloc verrà richiamato in automatico e la memoria sarà liberata.

Esiste anche un meccanismo di autorilascio, che può essere usato richiamando il metodo autorelease su un oggetto. Chiamando questo metodo al posto del metodo release, possiamo rimandare il rilascio dell'oggetto fino alla successiva esecuzione del ciclo degli eventi. Ciò significa che l'oggetto rimarrà in vita durante l'esecuzione corrente del codice, ma al termine dell'esecuzione del codice, quando l'applicazione tornerà in attesa l'oggetto sarà rilasciato decrementando il suo contatore. A quel punto se il retain count varrà 0 allora l'oggetto verrà deallocato.

Il framework Cocos2D nasconde agli sviluppatori questa gestione della memoria. Come abbiamo visto precedentemente per creare un nodo basta chiamare il metodo node della classe CCNode:

```
CCNode *myNode = [CCNode node];
```

Abbiamo creato un nodo senza richiamare né il metodo alloc né il metodo init. Questo perché il metodo node restituisce un oggetto già allocato e inizializzato:

```
+(id) node
{
       return [[[self alloc] init] autorelease];
}
```

Come vediamo il nodo è restituito lanciando su di esso il metodo autorelease. Questo significa che se non tratteniamo col metodo retain l'oggetto questo verrà deallocato. Tuttavia questo non avviene, perlomeno non avviene esplicitamente. Ogni qualvolta aggiungiamo un nodo alla scene graph invochiamo sul nodo parent il metodo addchild, questo metodo oltre ad ag-

giungere il nodo come figlio esegue su di esso anche un'operazione di retain. Quando invece rimuoviamo un figlio da un nodo parent, questo eseguirà sul nodo in questione una release. Cocos2D si occupa quindi di gestire al posto nostro la memoria, basterà costruire normalmente la scena aggiungendo i vari nodi come figli e le operazioni di retain e release saranno eseguite automaticamente dalle API offerte dal framework.

Il game loop in Cocos2D

Fino ad ora non abbiamo mai fatto riferimento al *game loop*, questo perché Cocos2D nasconde allo sviluppatore il meccanismo del game loop. Lo sviluppatore non deve far altro che implementare il proprio gioco attraverso i nodi e alla struttura ad albero delle scene. Sarà poi il game engine a gestire il game loop. In figura 4.5 vediamo com'è strutturato il game loop di

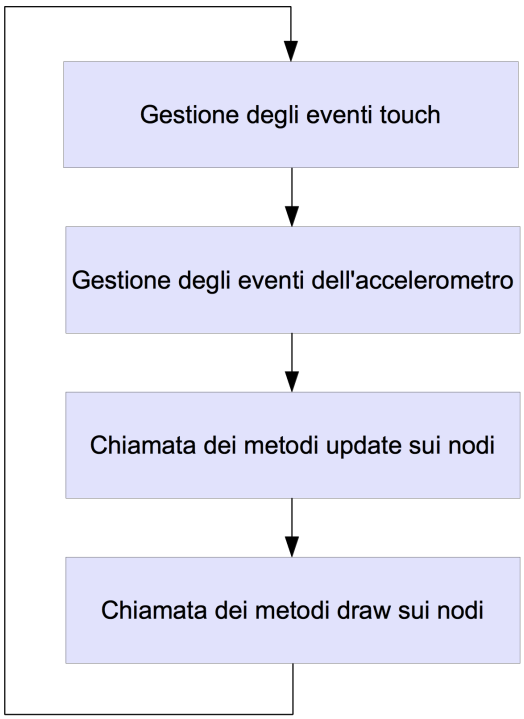

Figura 4.5: Struttura del game loop in Cocos2D

Cocos2D. Prima di tutto vengono gestiti gli eventi di input, poi vengono eseguiti i metodi update dei nodi che hanno attivato lo scheduling, infine viene disegnata la scena esguendo il metodo draw sui nodi che dispongono di una rappresentazione grafica.

Cocos2D si appoggia sulle librerie di Apple. Ogni qualvolta l'utente agisce sul touch screen: toccandolo, muovendo un dito o staccando il dito dallo schermo vengono generati una serie di eventi. Lo strato Cocoa Touch dell'architettura di iOS si occupa della loro gestione. Questi eventi sono ge-

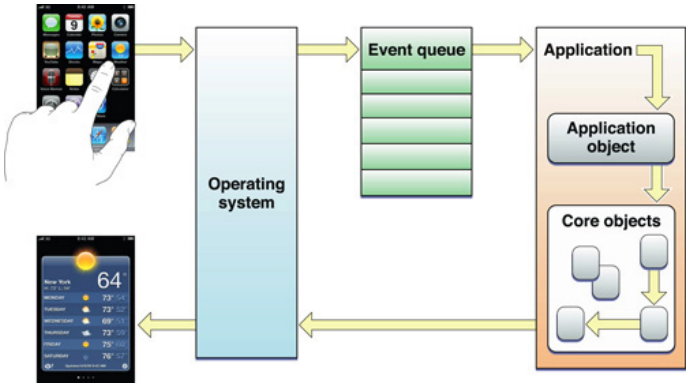

Figura 4.6: Event loop

stiti tramite un loop, mostrato in figura 4.6. Il sistema operativo cattura gli eventi di input e li infila in una coda. La coda viene affidata all'application object dell'applicazione, che procederà partendo dal primo evento a inviare gli eventi agli oggetti in grado di gestirli.

Gli eventi riguardanti il touch screen possono avvenire in qualunque momento e quando avvengono vengono infilati nella coda degli eventi. L'approccio con l'accelerometro e il giroscopio invece è diverso. Questi vegono interpellati dal sistema operativo ad intervalli regolari e le informazioni rilevate vengono infiliate nella coda degli eventi. Lo sviluppatore può decidere la frequenza con cui il sistema interroga l'accelerometro tramite il seguente codice:

```
UIAccelerometer* theAccelerometer = [UIAccelerometer
       sharedAccelerometer];
theAccelerometer.updateInterval = 1 / frequenza;
```

Tornando al game loop di Cocos2D vediamo come inizialmente si prenda a carico degli eventi del touch screen e dell'accelerometro affidando la loro gestione ai layer (o ai nodi) che hanno attivato la ricezione di questi input.

Una volta gestiti gli input, il game loop procede con la chiamata dei metodi update dei vari nodi. L'esecuzione dei metodi update avviene seguendo la priorità indicata quando si è attivata la chiamata programmata dei metodi. Se più metodi hanno lo stesso livello di priorità il game loop li eseguirà seguendo l'ordine dell'albero e lo z order. In questa fase il game loop scorre anche attraverso i metodi la cui chiamata è programmata con un determinato intervallo di tempo. Ogni qualvolta incontra uno di questi metodi controlla quanto tempo è passato dall'ultima volta che è stato eseguito, quindi se è passato l'intervallo di tempo indicato il metodo viene eseguito se no il game loop passa ad analizzare il metodo successivo.

Infine il game loop va a disegnare la scena sullo schermo. Scorrendo attraverso lo scene graph esegue il metodo draw sui nodi con una visualizzazione grafica. Per i nodi che si trovano allo stesso livello segue lo z order per l'esecuzione dei metodi draw. Procedendo dal nodo con valore di z order inferiore a quello con valore di z order maggiore.

Terminata la fase di disegno il game loop riprende dall'inizio con la gestione degli input.

4.3 Progettazione

Prima di approfondire la progettazione del gioco vero e proprio analizziamo la transizione tra le diverse scene dell'applicazione. Come abbiamo detto precedentemente in Cocos2D quando si sostituisce la scena visualizzata con una nuova scena il game engine carica in memoria la nuova scena e la manda in esecuzione, solo successivamente viene liberata la memoria occupata dalla vecchia scena. Questo può portare per un piccolo lasso di tempo ad un sovraccarico della memoria.

Una valida soluzione per aggirare questo problema può essere la realizzazione di una scena vuota, quindi che occupi poca memoria. Attivando su questa lo scheduling del metodo update, la prima volta che questo sarà eseguito la memoria della vecchia scena sarà già stata completamente liberata. Il caricamento della scena finale avviene proprio all'interno del metodo update. In questo modo in memoria non troveremo mai allocate contem-

poraneamente le due scene, ma solamente una di queste più la scena vuota che esegue la transizione.

4.3.1 La codifica degli input in comandi

Come detto precedente l'astronave è l'entità con cui si manifesta il giocatore all'interno del gioco. Questa non risponde direttamente agli input dell'utente, ma si limita a interpretare una serie di comandi. Occorre quindi introdurre un'entità che si occupi di ricevere gli input dell'utente e li codifichi in comandi. I comandi verranno poi captati dalle entità del gioco interessate.

Questa struttura in cui ci sono oggetti interessati alla ricezione di notifiche e oggetti che postano notifiche è la struttura tipica del *pattern observer*.

Il pattern observer

Il pattern observer è una valida soluzione nelle situazioni in cui uno o più oggetti (Observers) debbano essere informati riguardo ad una determinata notifica. Questo pattern fa sì che gli oggetti interessati alla notifica non conoscano l'oggetto che le genera. Gli observer si registrano presso il subject, segnalandogli così l'interesse di essere notificati quando avviene un certo evento, successivamente un oggetto segnala al subject l'avvenuta di un evento. Il subject notificherà tutti gli observer registrati precedentemente per quel determinato evento. Quando un observer non sarà più interessato ad un determinato evento lo segnalerà al subject cancellando la propria registrazione. Questa separazione fa sì che chi posta le notifiche non conosca chi le riceve e viceversa.

I framework di iOS includono la classe *NSNotificationCenter*, questa implementa il pattern observer. Un oggetto interessato a ricevere una certa notifica si registra presso il notificationcenter tramite il metodo:

```
- (void)addObserver:(id)notificationObserver selector:(SEL)
      notificationSelector name:(NSString *)notificationName
      object:(id)notificationSender
```

Il NotificationCenter spedisce notifiche sincrone agli observer registrati.

Per postare una notifica la classe mette a disposizione degli oggetti il metodo pubblico:

```
- (void)postNotificationName:(NSString *)notificationName object:(
      id)notificationSender userInfo:(NSDictionary *)userInfo
```

Il pattern observer è una soluzione perfetta per far sì che tutti gli oggetti interessati ad un determinato comando vengano notificati quando questo viene lanciato.

Per interagire con il videogioco l'utente ha a disposizione due tipi diversi di input:

- Inclinando il dispositivo il giocatore può muovere la navicella. L'inclinazione del dispositivo viene captata dal sensore accelerometro.
- Toccando lo schermo il giocatore può far sparare alla navicella un proiettile.

Come abbiamo visto le API di Cocos2D permettono alla classe *CCLayer* di ricevere gli input del *touch screen* e dell'*accelerometro*.

Definiamo una classe *InputLayer* che estenda la classe CCLayer e che abiliti la ricezione degli input quando viene inizializzata:

```
-(id) init
{
    if (self = [super init]) {
        //abilito l'input dall'accelerometro
        self.isAccelerometerEnabled = YES;
        //abilito l'input dal touch screen
        self.isTouchEnabled = YES;
    }
    return self;
}
```

Come abbiamo visto Cocos2D nella fase di gestione degli input del game loop se presenti eventi dell'accelerometro o del touch screen interpellerà questa classe eseguendo gli appositi metodi che gestiscono gli input. Vediamo l'implementazione del metodo per la gestione degli eventi dell'accelerometro:

```
-(void) accelerometer:(UIAccelerometer *)accelerometer
      didAccelerate:(UIAcceleration *)acceleration
{
    //modifico l'accelerazione portandola al formato dello schermo
          landscape
```

```
    CGPoint acceleration_ = CGPointMake(- acceleration.y,
           acceleration.x);
    //converto l'accelerazione in una stringa perché la notifica
           accetta solo degli oggetti
    NSString *accelerationString = NSStringFromCGPoint(
           acceleration_);

    //genero e posto la notifica che muoverà l'astronave
    NSDictionary *dict = [NSDictionary dictionaryWithObject:
           accelerationString forKey:@"accelerationString"];
    [[NSNotificationCenter defaultCenter] postNotificationName:@"
           SET_SHIP_ACCELERATION" object:self userInfo:dict];
}
```

In questo caso la notifica deve comunicare anche le informazioni dell'accelerometro, queste per essere postate come notifiche devono essere nel formato di oggetto, quindi vengono prima convertite in una stringa dal formato CGPoint. Infine posta la notifica sul NotificationCenter.

Per gli input del touch screen abbiamo visto che possiamo avere tre diversi eventi, ma nel nostro caso siamo interessati a captare solo quelli che riguardano l'inizio di un nuovo tocco, in tal caso l'astronave dovrà sparare un proiettile. Non abbiamo interesse a gestire gli eventi che segnalano il movimento delle dita sullo schermo e la fine di un tocco in quanto il gioco non deve rispondere a queste situazioni.

```
-(void) ccTouchesBegan:(NSSet*)touches withEvent:(UIEvent*)event
{
    //quando tocco lo schermo genero l'evento che fa sparare l'
           astronave
    [[NSNotificationCenter defaultCenter] postNotificationName:@"
           SHIP_SHOOT" object:self];
}
```

Come vediamo nello stralcio di codice qui sopra quando lo schermo viene toccato il layer genera il comando che fa sparare l'astronave e lo posta sul NotificationCenter.

Gli oggetti che si sono registrati precedentemente al NotificationCenter per le notifiche indicate qui sopra, saranno quindi notificati.

4.3.2 Astronave, proiettili e asteroidi

Analizziamo ora le entità che percepirà direttamente l'utente, cioè le entità con una rappresentazione grafica. Queste entità compongono la scena principale del gioco e sono: astronave, proiettili e asteroidi.

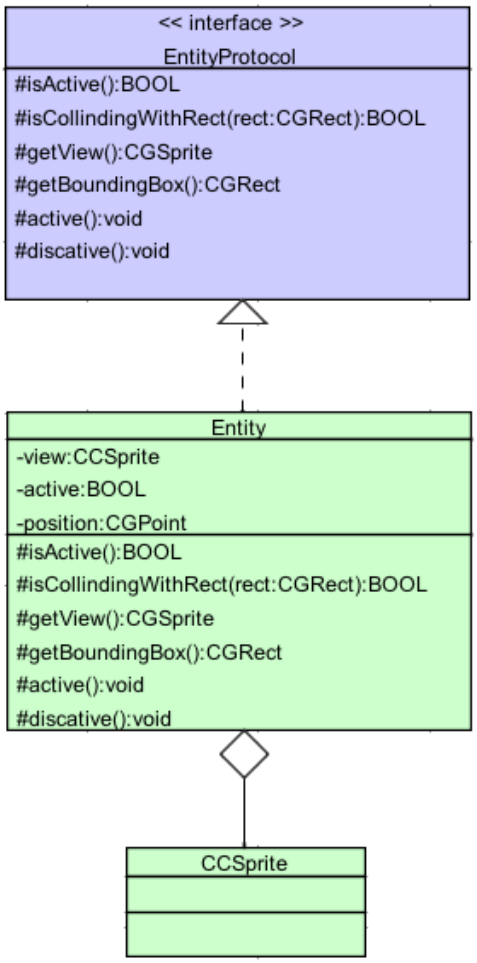

Figura 4.7: Modello UML di un'entità generica

In figura 4.7 è mostrato il modello UML di un'entità generica. Come vediamo dal modello, un'entità racchiude al suo interno la sua visualizzazione in forma di CCSprite. Il legame tra l'entità e la sua vista è in generale

di *aggregazione*, successivamente verrà chiarito il perché di questa scelta. Oltre alla view, tra gli attributi della classe troviamo anche un booleano che indica se l'entità è attiva o meno e la posizione dell'entità.

Un'entità quando è attiva ha la propria view visibile ed ha attivato lo scheduling del metodo *update*, quando invece è disattiva la view è invisibile e lo scheduling di tutti i metodi è disattivato. Vediamo l'implementazione del metodo active:

```
-(void) active
{
    active = YES; //setto a yes il flag
    view.visible = YES; //rendo visibile la sua vista
    [self scheduleUpdate]; //attivo lo scheduling del metodo update
}
```

Successivamente vedremo che ciascuna entità specifica (astronave, proiettile, asteroide) implementerà il proprio metodo update. Non riportiamo l'implementazione del metodo disactive in quanto fa l'esatto contrario del metodo active.

Oltre ai metodi per ottenere gli attributi dell'entità e ai metodi per attivarla e disattivarla, troviamo:

- il metodo getBoundingBox che restituisce le coordinate del rettangolo che contiene la vista dell'entità

- il metodo isCollidingWithRect che accettando in ingresso le coordinate di un rettangolo, restituisce un booleano che indica se l'entità è in collisione o meno col rettangolo. Questo metodo effettua il controllo, verificando se ci sono intersezioni tra il rettangolo ricevuto in input e il rettangolo della view.

La classe Entity dovrebbe essere una classe astratta in quanto non implementa il metodo update, ma purtroppo l'Objective-C essendo un linguaggio datato non dispone di questa funzionalità.

L'astronave

Per la gestione dell'astronave si è creata l'interfaccia (cioè il protocollo in Objective-C) ShipProtocol, che estende l'interfaccia EntityProtocol vista

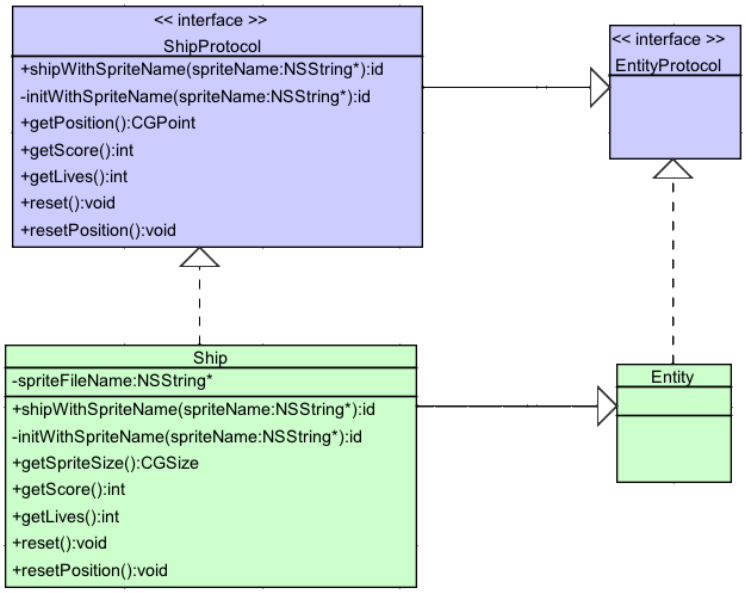

Figura 4.8: Modello UML della classe Ship

in figura 4.7. Come vediamo in figura 4.8 la classe *Ship* implementa il protocollo ShipProtocol ed estende la classe Entity.

La classe Ship rappresenta l'astronave, quindi il giocatore, dovrà perciò avere tra attributi il numero di vite rimaste (lives) e i punti totalizzati (points).

Tra i metodi analizziamo il costruttore che come vediamo come parametro d'ingresso riceve il nome dell'immagine da caricare nello sprite:

```
+(id) shipWithSpriteName:(NSString*) spriteName
{
    return [[[self alloc] initWithSpriteName:(NSString*) spriteName
           ] autorelease];
}
```

Come possiamo vedere il costruttore si attiene al modello di memoria di Cocos2D, restituisce infatti un'oggetto già inizializzato e in autorelease. Come vedremo successivamente affinché la memoria dell'oggetto non venga subito deallocata lo aggiungeremo all'albero dello scene graph.

Analizziamo l'implementazione del metodo per inizializzare l'oggetto:

```
-(id) initWithSpriteName:(NSString*) spriteName
{
    if (self = [super init]) {
        spriteFileName = spriteName;
        numberOfFrames = 5;
        delay = 0.04f;
        initialLives = 5;     //setto a 5 il numero di vite iniziali
        lives = initialLives;
        score = 0;       //setto il punteggio a 0
        acceleration = CGPointMake(0, 0); //setto accelerazione
               inizialmente a 0
        winSize = [[CCDirector sharedDirector] winSize]; //recupero
                le dimensioni dello schermo

        // creo un animation con tutti i frames
        CCAnimation* anim = [CCAnimation animationWithFrame:
               spriteFileName frameCount:numberOfFrames delay:delay
               ];
        //applico l animation allo sprite
        CCAnimate* animate = [CCAnimate actionWithAnimation:anim];
        CCRepeatForever* repeat = [CCRepeatForever actionWithAction
               :animate];
        [view runAction:repeat];

        //calcolo i margini entro cui si può muovere l'astronave
        {...}

        //posiziono la navicella inizialmente a sinistra a metà
               schermo
        position = CGPointMake(leftBorderLimit, winSize.height *
               0.5f);
        view.position = position;

        [self addChild: view];   //aggiungo all'entità la view come
                figlia
        [self active];      //attivo l'entità

        //inscrivo la navicella alla ricezione delle notifiche che
               la riguardano
```

```
        //cambio di accelerazione
        [[NSNotificationCenter defaultCenter] addObserver:self
               selector:@selector(setAcceleration:) name:@"
               SET_SHIP_ACCELERATION" object:nil];
        //comando per far sparare la navicella
        [[NSNotificationCenter defaultCenter] addObserver:self
               selector:@selector(shoot:) name:@"SHIP_SHOOT" object
               :nil];
        //collisione con un asteroide
        [[NSNotificationCenter defaultCenter] addObserver:self
               selector:@selector(gotHit:) name:@"
               SHIP_COLLIDE_WITH_ASTEROID" object:nil];
        //asteroide colpito da un proiettile
        [[NSNotificationCenter defaultCenter] addObserver:self
               selector:@selector(incrementScore:) name:@"
               BULLET_COLLIDE_WITH_ASTEROID" object:nil];
    }
    return self;
}
```

Come vediamo qui sopra la view (cioè la rappresentazione grafica dell'entità) è aggiunta come figlia all'entità stessa. In questo caso quindi il legame diventa più forte di una semplice aggregazione, diventa una *composizione*. L'oggetto iscrive se stesso alla ricezione di una serie di notifiche presso il NotificationCenter, analizzamo queste notifiche:

- SET_SHIP_ACCELERATION: questa notifica segnala all'astronave che la sua accelerazione è cambiata. Come abbiamo visto prima, questa notifica è inviata dall'InputLayer quando rileva un cambio di stato dell'accelerometro.

- SHIP_SHOOT: questa notifica segnala all'astronave che deve sparare. Quando viene notificata l'oggetto chiama il metodo shoot che invia una nuova notifica includendo la posizione dell'astronave.

- SHIP_COLLIDE_WITH_ASTEROID: segnala che l'astronave si è scontrata con un asteroide, verranno quindi decrementate il numero di vite e se le vite sono finite verrà inviata una notifica che segnala lo stato di game over.

- BULLET_COLLIDE_WITH_ASTEROID: segnala che un proiettile ha colpito l'asteroide, il punteggio del giocatore verrà perciò incrementato.

Alla fine dell'inizializzazione l'entità viene attivata, rendendola così visibile e attivando su di essa lo scheduling del metodo update. Riportiamo qui sotto le parti principali:

```
-(void) update:(ccTime)delta
{
    //calcolo la nuova posizione in base all'accelerazione
    CGPoint newPosition = CGPointMake(position.x + (0.5f *
           acceleration.x * delta * delta), position.y + (0.5f *
           acceleration.y * delta * delta));

    //controllo che la nuova posizione sia entro i limiti dello
           schermo
    {...}

    position = newPosition;  //setto la nuova posizione allo sprite
    view.position = position;
}
```

Nel metodo update è implementato il codice che fa muovere l'astronave in base alla sua accelerazione (che proviene dall'accelerometro). Questo metodo viene chiamato ad ogni frame e riceve come parametro delta: il tempo trascorso dall'ultima volta che il metodo è stato chiamato. Si può notare che il calcolo della nuova posizione dell'astronave segue l'equazione del *moto accelerato*.

Gli asteroidi

La classe *Asteroid* definisce il modello di un asteroide, estende la classe Entity ed implementa il protocollo *AsteroidProtocol*.

La classe Asteroid implementa il metodo *launch* che permette di lanciarli. Questo metodo prende come input la velocità dell'asteroide, la posizione e scale. Il parametro scale è un fattore moltiplicativo che indica di quanto sarà scalato lo sprite dell'asteroide, permette quindi di regolarne la dimensione. Quando questo metodo viene eseguito l'asteroide viene attivato, attivando così su di lui lo scheduling del metodo update. Il metodo update non sarà

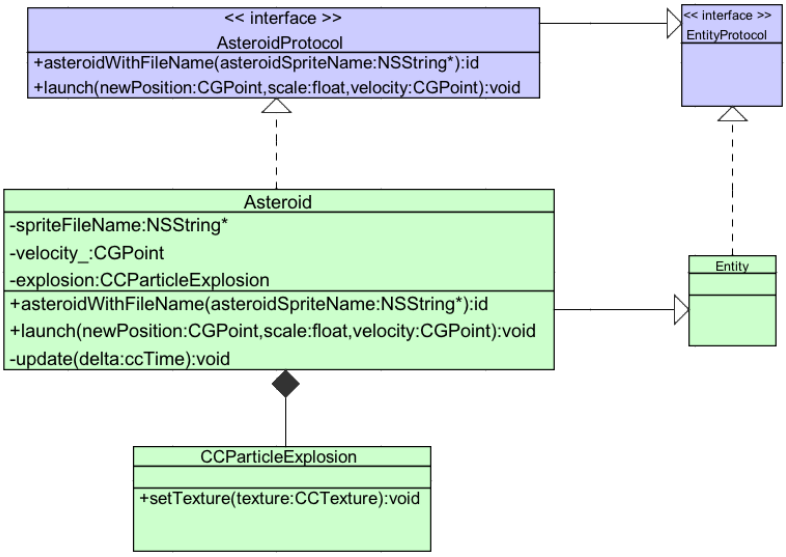

Figura 4.9: Modello UML della classe Asteroid

poi tanto diverso dal metodo update dell'astronave, con la differenza che farà un controllo sulla posizione dell'asteroide, e se questo si troverà fuori dallo schermo lo disattiverà.

Gli asteroidi quando colpiscono l'astronave o quando vengono colpiti da un proiettile devono esplodere, per questo la classe Asteroid implementa il metodo pubblico *explode*. Questo metodo disattiva l'asteroide e mostra sullo schermo un'esplosione. Per creare l'effetto dell'esplosione Cocos2D mette a disposizione la classe *CCParticleExplosion*. Questa classe permette di istanziare oggetti che riproducono l'esplosione utilizzando una piccola immagine che animano e ripetono tante volte sullo schermo, al fine di ottenere un effetto come quello mostrato in figura 4.1. Ogni asteroidi è quindi composto tramite un legame di composizione da un oggetto explosion che è istanza di questa classe.

Durante il gioco possono esserci più asteroidi sulla scena contemporaneamente, e ognuno di questi può muoversi con una velocità diversa dagli altri. La soluzione più semplice per gestirli è quella di creare un'asteroide nel momento in cui deve essere lanciato e eliminare dalla memoria l'ateroide, quando esplode o quando esce "illeso" dallo schermo. Seguendo questo

semplice ciclo di vita, dovremmo allocare un nuovo asteroide ogni volta che vogliamo lanciarlo. Il continuo allocare e deallocare la memoria di un astoroide può rallentare notevolmente il flusso di esecuzione del gioco. La soluzione di allocare dinamicamente un nuovo asteroide ogni volta che lo si vuole lanciare non è quindi la soluzione migliore.

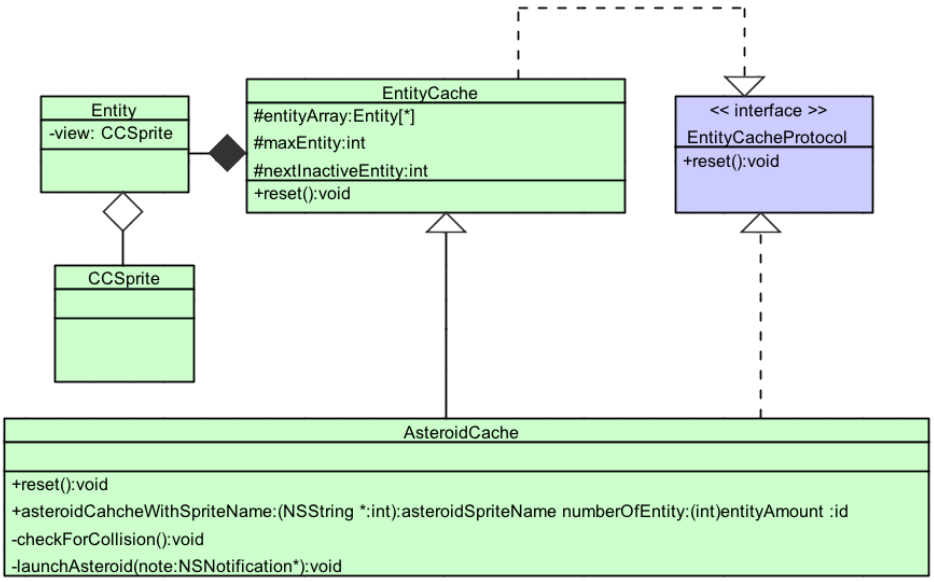

Figura 4.10: Modello UML della classe AsteroidCache

Introduciamo un'entità chiamata *EntityCache* che organizza una serie di oggetti Entity in un array di lunghezza indefinita denominato entityArray, vedi figura 4.10. Tra gli attributi di questa classe troviamo il valore intero *nextInactiveEntity*, che indica nell'array la posizione della prossima Entity disattiva. Implementa inoltre il metodo *reset* che disattiva tutte le entità dell'array. Questa classe dovrebbe essere astratta, ma il linguaggio Objective-C non permette questo costrutto.

AsteroidCache è la classe che nello specifico permette di gestire gli asteroidi, questa classe estende la classe EntityCache. Il costrtuttore, che si attiene al modello di memoria usato da Cocos2D, accetta come parametro d'ingresso il nome del file con l'immagine dell'asteroide e il numero di asteroidi da allocare nell'array. Riportiamo l'implementazione:

```
+(id) asteroidCahcheWithSpriteName:(NSString *)asteroidSpriteName
       numberOfEntity:(int)entityAmount
{
    return [[[self alloc] initWithAsteroidSpriteName:
           asteroidSpriteName numberOfEntity:entityAmount]
           autorelease];
}

-(id) initWithAsteroidSpriteName:(NSString *)asteroidSpriteName
       numberOfEntity:(int)entityAmount
{
    if (self = [super init]) {
        //numero massimo di asteroidi allocabili contemporaneamente
        maxEntity = entityAmount;

        //inizializzo l'array per contenere gli asteroidi
        entityArray = [[CCArray alloc] initWithCapacity:maxEntity];

        // creo il batchnode che poi conterrà gli sprite
        CCSpriteFrame* asteroidFrame = [[CCSpriteFrameCache
               sharedSpriteFrameCache] spriteFrameByName:
               asteroidSpriteName];
        CCSpriteBatchNode *asteroidSpriteBatch = [CCSpriteBatchNode
                batchNodeWithTexture:asteroidFrame.texture];

        // riempo l'array
        for (int i=0; i < maxEntity; i++) {
            //creo un asteroide con lo sprite
            CCNode<AsteroidProtocol> *asteroid = [Asteroid
                   asteroidWithFileName:asteroidSpriteName];
            [entityArray addObject:asteroid]; //aggiungo l'asteroide
                    all'array
            [self addChild:asteroid]; //aggiungo l'asteroide al nodo
                    come figlio
            //aggiungo lo sprite dell'asteroide allo spite batch
            [asteroidSpriteBatch addChild:[asteroid getView]];
        }

        nextInactiveEntity = 0; //azzero il counter degli asteroidi
```

```
            attivi
        [self addChild:asteroidSpriteBatch]; //aggiungo lo sprite
            batch al nodo
    }

    //iscrivo l'oggetto al comando che lancia un asteroide
    [[NSNotificationCenter defaultCenter] addObserver:self selector
        :@selector(launchAsteroid:) name:@"LAUNCH_ASTEROID"
        object:nil];
    //avvio lo scheduling del metodo update (dove si farà il
        controllo delle collisioni)
    [self scheduleUpdate];

    return self;
}
```

Come si vede nell'implementazione, viene creato l'array contenente tutti gli asteroidi e l'oggetto iscrive se stesso alle notifiche che segnalano quando si vuole lanciare un asteroide. Quando verrà notificato lancerà il metodo *launchAsteroid* che eseguirà sul primo asteroide disattivo dell'array il metodo launch per lanciarlo e incrementerà nextInactiveAsteroid. Se dopo l'incremento nextInactiveAsteroid equivale alla lunghezza dell'array, questo viene riportato a 0. È quindi importante allocare una quantità non troppo piccola di asteroidi, affinché non si vada a rilanciare un asteroide che non ha ancora concluso il proprio percorso.

Si può notare che all'interno dell'implementazione viene creato un oggetto dalla classe SpriteBatchNode. Questa classe è un nodo, quando si crea questo nodo si specifica il file di un'immagine, successivamente il nodo accetta come figli tutti i CCSprite la cui immagine è quella indicata precedentemente nella creazione del nodo batch. Cocos2D quando in fase di draw incontra un nodo SpriteBatchNode saprà che tutti i suoi figli sono sprite che mostrano la stessa immagine, potrà quindi disegnarli sullo schermo con un'unica passata. L'utilizzo di questa classe permette di ottimizzare notevolmente la fase di draw.

Nella creazione di un'oggetto AsteroidCache aggiungiamo ad un nodo batch tutti gli sprite degli asteroidi allocati, così da ottimizzare le performance del gioco. Per questo motivo la relazione che lega la classe Asteroid con la sua view che è istanza della Classe CCSprite è di semplice aggregazione.

Il controllo delle collisioni

La classe AsteroidCache implementa anche il metodo *checkForCollision*. Questo metodo ha il compito di controllare se gli asteroidi entrano in collisione con l'astronave o con dei proiettili.

Per controllare se ci sono collisioni con l'astronave, essendo l'astronave un'istanza della classe Entity (o sua derivata) basterà invocare su di questa il metodo isCollidingWithRect passandogli come parametro la bounding box di ciascun asteroide attivo. Se riceverà un valore positivo farà esplodere l'asteroide e posterà sul NotificationCenter la notifica che segnala la collisione tra l'asteroide e l'astronave. Il NotificationCenter, notificherà quindi gli iscritti, tra questi troverà sicuramente l'astronave, che decrementerà così il proprio numero di vite.

Per controllare se ci sono collisioni con i proiettili la questione è più complicata. Possiamo infatti avere più proiettili contemporaneamente nella scena di gioco. Per questo motivo può essere utile anche per i proiettili introdurre un oggetto simile all'AsteroidCache che permette di gestire tutti i proiettili del gioco, come vedremo successivamente chiameremo questo "raccoglitore" BulletCache. Per ora limitiamoci a dire che questa classe BulletCache dovrà implementare un metodo che chiameremo areBulletsCollidingWithRect che ricevendo come input le coordinate di un rettangolo (cioè una bouding box) restituisca un booleano che indichi se esiste un proiettile che in quel momento sta collidendo con quel rettangolo.

La classe AsteroidCache per verificare se ci sono collisioni con i proiettili può attraverso un riferimento al BulletCache invocare su di esso il metodo areBulletsCollidingWithRect passandogli come parametro la bounding box di ciascun asteroide attivo. Ogni qualvolta individuerà una collisione, posterà sul NotificationCenter una notifica per sengnalarla, questa verrà sicuramente reindirizzata all'astronave che incrementerà il punteggio del giocatore.

Ogni qualvolta individuerà una collisione di un asteroide con l'astronave o con un proiettile invocherà sull'asteroide in questione il metodo explode, che lo farà esplodere e lo disattiverà.

I proiettili

Come abbiamo già accennato la gestione dei proiettili non sarà tanto differente da quella degli asteroidi. Introdurremo una classe *BulletCache* che

estendendo la classe *EntityCache* gestirà tutti i proiettili del gioco mettendoli in un array. Questa classe si scriverà al NotificationCenter alla notifica che segnala il lancio di un proiettile. Ogniqualvolta sarà notificato lancerà il primo proiettile inattivo dell'array.

Come visto nella sezione precedente, la classe dovrà implementare il metodo areBulletsCollidingWithRect che riportiamo qui sotto:

```
-(bool) areBulletsCollidingWithRect:(CGRect)rect
{
	bool isColliding = NO;
	//controllo tutti i proiettili dell'array
	for (int i=0; i < maxEntity; i++)
	{
		CCNode<BulletProtocol> *bullet = (CCNode<
			BulletProtocol> *) [entityArray objectAtIndex
			:i];
		//eseguo il controllo solo se il proiettile è attivo
		if ([bullet isActive])
		{
			//controllo se c'è un'intersezione
			if ([bullet isCollidingWithRect: rect])
			{
				isColliding = YES;
				//disattivo il proiettile
				[bullet disactive];
				break;
			}
		}
	}
	return isColliding;
}
```

Vediamo nell'implementazione come ogni qualvolta venga individuato un proiettile che collide questo viene disattivato, facendolo così sparire dallo schermo e disattivandogli lo scheduling del metodo update.

4.3.3 La scena di gioco

Chiamiamo scena di gioco, quell'entità che contiene tutti gli elementi che compongono il gioco. La scena di gioco non si limiterà quindi a contenere i soli elementi con una rappresentazione grafica, ma conterrà anche tutti gli elementi che permetto di organizzare le varie entità. La scena di gioco dovrà quindi racchiudere tutte le classi viste finora.

Nell'analisi delle varie entità abbiamo visto che la classe AsteroidCache esegue il controllo sulle collisioni degli asteroidi con le altre entità della scena. Per fare ciò la classe ha bisogno di un riferimento all'oggetto che individua l'astronave e alla classe che gestisce i proiettili. La soluzione più semplice potrebbe essere quella di creare l'oggetto AsteroidCache solo dopo aver creato tutte le altre classi e passargli in fase di costruzione i riferimenti ad esse. Questa soluzione sebbene sia molto semplice è decisamente poco elegante.

In alternativa possiamo far sì che siano le entità interessate come ad esempio l'AsteroidCache ad interrogare la scena di gioco per ottenere i riferimenti agli altri oggetti. Per fare ciò è necessario che la classe AsteroidCache conservi un riferimento alla scena di gioco. Anche questa soluzione è poco elegante.

Visto che di scene in esecuzione possiamo averne solo una attiva alla volta, possiamo pensare ad implementare la scena di gioco seguendo il pattern *singleton*. Il pattern singleton è una soluzione ottimale per il nostro problema visto che oltre a far sì che non possano esistere più istanze della stessa classe nello stesso momento offre anche un punto globale di accesso.

Implementando sulla scena di gioco anche una serie di metodi che restituiscono le diverse entità che la compongono (vedi figura 4.11), gli oggetti interessati come ad esempio l'AsteroidCache possono ottenere il riferimento alle altre entità per eseguire il controllo delle collisioni.

Il costruttore della classe GameScene restituisce un oggetto CCScene, questo oggetto è la scena principale del gioco e una volta mandata in esecuzione sul CCDirector il gioco inizia.

Nella fase di inizializzazione tutti i nodi che compongono il gioco vengono aggiunti alla scena, non solo i nodi con una rappresentazione grafica ma anche i nodi come l'IputLayer che traduce gli input in comandi o la classe che gestisce il livello di gioco. Ogni oggetto viene aggiunto come figlio della scena, seguendo la struttura su cui si basa Cocos2D. È importante che

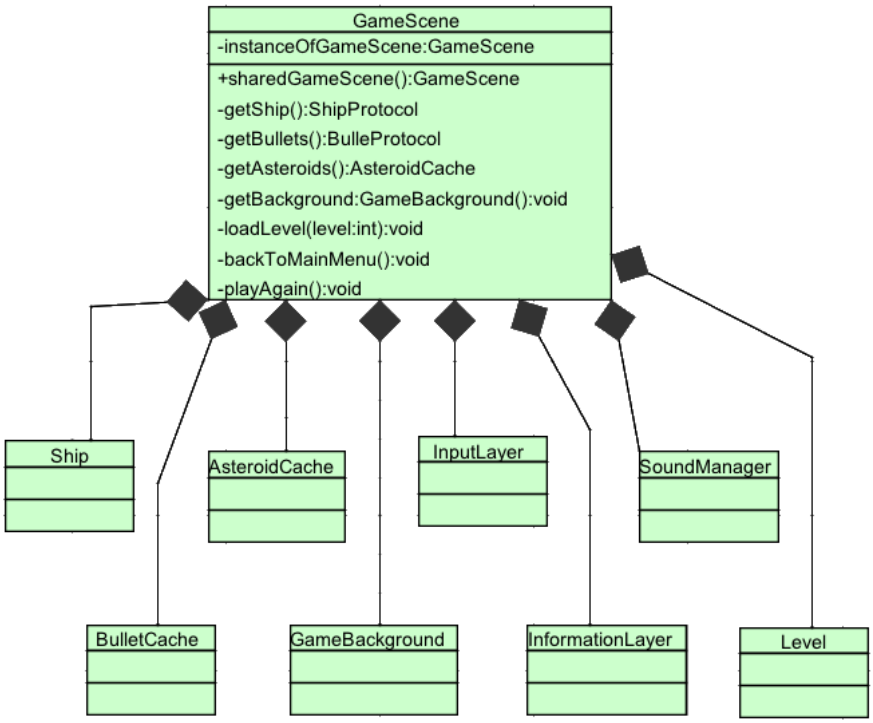

Figura 4.11: Modello UML semplificato della classe GameScene

ciascun nodo venga aggiunto con un tag univoco. Solo così sarà possibile recuperare un determinato nodo quando sarà evocato il metodo che lo richiede, ad esempio getShip. Per la gestione dei tag sì è deciso di usare un enumerato.

La classe GameScene è iscritta al NotificationCenter per la notifica di *game over*. In caso di game over interromperà il gioco resettando attraverso gli appositi metodi le entità asteroidi e proieittili. Infine mostra una schermata con il messaggio di game over e il punteggio totalizzato.

Nell'immediato verrebbe da pensare alla schermata di game over come ad una nuova scena. Questa però non è la scelta migliore, caricando la nuova scena verrebbe liberata la memoria della scena precedente, cioè quella del gioco. Nel caso in cui il giocatore desiderasse ricominciare immediatamente una nuova partita dovremmo caricare nuovamente l'intera scena di gioco. Per evitare questa inutile situazione il messaggio di game over viene mo-

strato tramite un layer (con sfondo nero) che si sovrappone a tutti gli altri nodi della scena di gioco. Con questa soluzione il giocatore se lo desidera può ricominciare immediatamente a giocare senza attendere nuovamente il caricamento degli elementi di gioco.

Come si vede in figura 4.11 la scena di gioco è costituita anche da altri componenti di cui non si è discusso. Sono principalmente componenti semplici che inseriscono degli elementi di corredo allo scheletro principale di gioco. Tra questi possiamo notare l'InformationLayer che mostra sullo schermo le vite rimaste e il punteggio del gocatore. La classe SoundManager gestisce i suoni e in fase di inizializzazione si iscrive ad una serie di notifiche (come ad esempio il lancio di un proiettile) presso il NotificationCenter.

Vale la pena citare la classe *Level*, questa classe implementa un livello del gioco. Abbiamo visto come il nostro sistema si basi molto sugli eventi e le notifiche. Tra le varie notifiche troviamo quella per lanciare gli asteroidi. La classe Level che dietro potrà nascondere anche meccanismi di intelligenza artificiale può captare i vari eventi del gioco, come ad esempio i comandi che riceve l'astronave o le eventuali collisioni e valutare di conseguenza come mettere in difficoltà o meno il giocatore. Ogni qualvolta che vorrà lanciare un asteroide le basterà semplicemente postare una notifica di lancio sul NotificationCenter.

4.3.4 Interazioni

Ora che abbiamo definito quali sono i principali elementi che compongono lo scheletro del videogioco riepiloghiamo come interagiscono tra loro. Per questione di spazi non riporteremo un sequency diagram completo. Ci limiteremo a riportare i sequency diagrams di alcuni dei casi più ricorrenti all'interno del videogioco.

In figuara 4.12 vediamo la sequenza di azioni che si susseguono quando l'InputLayer rileva un cambio nell'accelerazione. Precedentemente l'astronave, rappresentata dalla classe Ship si è registrata presso il NotificationCenter segnalandogli di voler essere notificata quando viene postata una notifica SET_ACCELERATION. L'InputLayer quando riceve un segnale dall'accelerometro posta questa notifica sul NotificationCenter, il quale provvederà a notificare tutti gli oggetti registrati precedentemente, nel nostro caso solo l'astronave. Viene eseguito il metodo setAcceleration della classe Ship passandogli come parametro un NSDictionary che contiene il

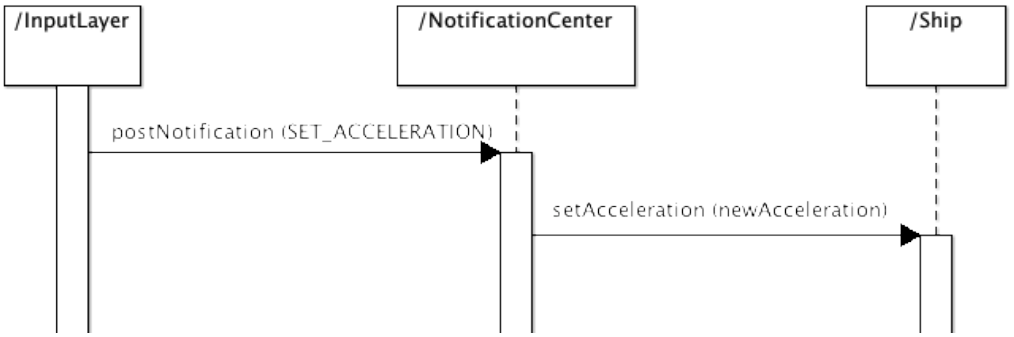

Figura 4.12: Sequency diagram per settare l'accelerazione dell'astronave

valore della nuova accelerazione. La classe Ship salverà questo valore e alla prossima chiamata del metodo update, calcolerà e setterà la nuova posizione dell'astronave utilizzando il valore di accelerazione ricevuto.

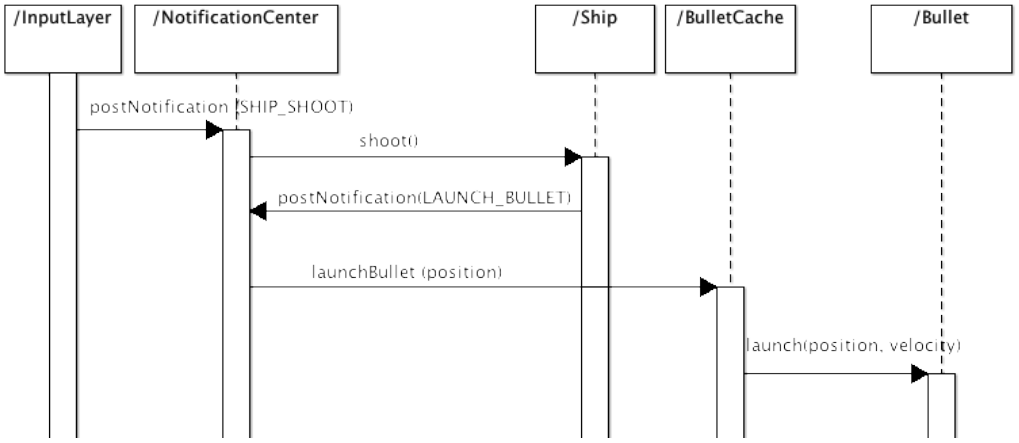

Figura 4.13: Sequency diagram del lancio di un proiettile

In figura 4.13 vediamo la sequenza di azioni che si susseguono quando l'InputLayer rileva un tocco nel touch screen. Il segnale di input dello schermo viene codificato in un comando che farà sparare l'astronave. Anche in questo caso non viene mostrata la registrazione da parte dell'astronave e del BulletCache alle rispettive notifiche. Il NotificationCenter quando riceve la notifica da parte dell'InputLayer, contatta tutti gli oggetti che

si erano registrati precedentemente alla notifica SHIP_SHOOT, nel nostro caso solo la classe Ship. Viene invocato il metodo shoot sull'astronave. Questo metodo non fa altro che postare una notifica LAUNCH_BULLET sul NotificationCenter. Questa notifica include come parametro la posizione attuale dell'astronave.

Il NotificationCenter invocherà sugli oggetti registrati alla notifica il metodo da loro indicato in fase di registrazione, nel nostro caso il metodo launchBullet sull'oggetto BulletCache. Dentro al metodo launchBullet, che riceve come parametro la posizione dell'astronave, verrà individuato il primo proiettile disattivato dell'array e verrà chiamato su di esso il metodo launch che sparerà finalmente il proiettile. Il metodo launch riceve come parametri d'ingresso la posizione da cui partirà il proiettile (cioè la posizione dell'astronave) e la velocità con cui viaggerà il proiettile. Questa velocità è un attributo della classe BulletCache.

Analizziamo il perché abbiamo implementato questo sistema utilizzando due diverse notifiche. L'InputLayer non conosce la posizione dell'astronave e non è compito suo recuperarla. In alternativa il BulletCache ogni volta che deve lanciare un proiettile avrebbe potuto recuperare tramite un riferimento all'oggetto Ship la posizione dell'astronave. Questa soluzione sarebbe stata poco elegante in quanto la posizione dell'astronave non ha niente a che fare con il BulletCache.

La prima notifica segnalerà all'astronave che è stato ricevuto il comando per farla sparare. L'astronave creerà una nuova notifica contenente la sua posizione (cioè la posizione di partenza del proiettile) che sarà poi ricevuta dal BulletCache che conoscerà così il punto da cui lanciare il proiettile.

La doppia notifica nel lancio di un proiettile, non è una soluzione strettamente necessaria, ma è la soluzione migliore in quanto permette di tenere separato i compiti di ciascun elemento. Inoltre garantisce una grossa flessibilità e modularità per gli sviluppi futuri.

Infine in figura 4.14 vediamo la sequenza di azioni che si susseguono per lanciare un asteroide all'interno del gioco. La classe AsteroidCache si è iscritta precedentemente presso il NotificationCenter alla notifica LAUNCH_ASTEROID. La classe Level, che gestisce l'avanzamento del livello del gioco quando opportuno invierà questa notifica al NotificationCenter. La notifica conterrà la posizione di partenza, la velocità e la dimensione dell'asteroide da lanciare.

Il NotificationCenter avvisa quindi l'AsteroidCache eseguendo il meto-

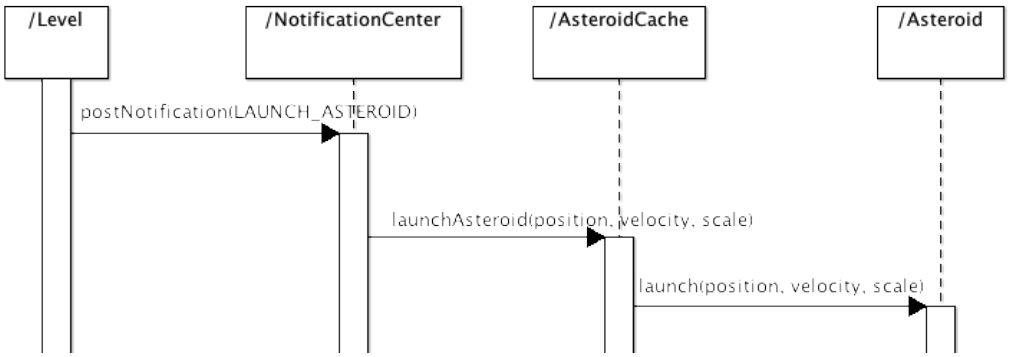

Figura 4.14: Sequency diagram del lancio di un asteroide

do da lui indicato in fase di registrazione, cioè il metodo launchAsteroid. L'AstroidCache individuerà il primo asteroide inattivo dell'array di asteroidi precedentemente allocati ed eseguirà su questo il metodo launch che lancerà infine l'asteroide.

Si è deciso di utilizzare il sistema di notifiche anche per il lancio degli asteroidi così da lasciare il sistema aperto a sviluppi futuri. In futuro volendo sarà possibile implementare un agente AI che captando le notifiche che regolano le azioni dell'astronave possa lanciare di conseguenza in qualunque momento un asteroide nella posizione, con la velocità e la dimensione desiderata.

4.4 Considerazioni

Il gioco realizzato sebbene sia semplice riesce a gestire i diversi input dell'accelerometro e del touch screen offrendo così al giocatore un'interfaccia intuitiva e semplice da usare.

La prova del gioco su un dispositivo iPhone 3GS dotato della versione 5.0.1 del sistema operativo iOS ha mostrato come il gioco scorra fluido e come la risposta ai comandi sia immediata. Sono stati svolti una serie di test attraverso Instruments, uno strumento offerto da Apple per monitorare in tempo reale le performance dell'applicazione. Da questi test è emerso che il consumo di memoria da parte del gioco è di poco più di 6 mb. Considerando che le immagini usate dal videogioco pesano circa 100 kb e che le risorse

audio pesano circa 5,5 mb, la porzione di memoria usata complessivamente dal gioco è da ritenersi molto ridotta. Sempre tramite Instruments abbiamo osservato come il valore fps cioè il numero di frame per secondo sia sempre su 60 con ogni tanto qualche rara caduta di 1 o 2 punti. Anche questo risultato è da ritenere molto buono in quanto il refresh massimo dello schermo del telefono è proprio di 60 fps.

Il sistema realizzato per la rilevazione delle collisioni risulta essere un po' limitato. Basando il controllo sulla semplice ricerca di intersezioni tra le bounding box di due o più unità, capita che vengano segnalate collisioni anche quando in realtà le due immagini delle entità non si stanno toccando. Nella maggior parte delle occasioni questo problema risulta essere quasi impercettibile. Tuttavia nel controllo delle collisioni tra l'astronave e asteroidi molto grandi il problema si nota più facilmente. Per risolverlo si potrebbe definire per ciascuna entità una struttura che contenga i vertici del bordo dell'immagine che la rappresenta e realizzare un sistema di collisioni che basi il controllo sulle informazioni contenute in queste strutture e non più sulle semplici bounding box. In alternativa si potrebbe introdurre un physic engine come Box2D che comprende già questi strumenti.

Facendo largo uso del pattern observer per lo scambio di messaggi tra le diverse parti del gioco abbiamo mantenuto il gioco aperto a sviluppi futuri. Sicuramente sarebbe interessante introdurre nel gioco un agente dotato di intelligenza il quale iscrivendo se stesso alle notifiche che indicano i comandi dell'utente, possa mettere in difficoltà il giocatore cercando di prevedere le sue mosse future.

Capitolo 5

Conclusioni

Attraverso lo studio delle architetture e dei sensori integrati negli smartphones abbiamo capito come il mondo dei videogiochi si sia trasformato. Si è visto come gli sviluppatori possano sfruttare questi sensori per realizzare interfacce di gioco più intuitive, più trasparenti, attraendo così utenze rimaste fino ad ora estranee al mondo dei videogames. Abbiamo approfondito l'architettura ARM e come questa grazie alla modularità e ai bassi consumi sia riuscita a diffondersi in quasi tutti i dispositivi oggi in commercio. Si sono inoltre analizzate le principali piattaforme software degli smartphones, studiando come queste siano strutturate e che strumenti mettano a disposizione dello sviluppatore.

Analizzando la struttura tipica di un team di sviluppo di videogiochi abbiamo compreso quali siano le principali figure che lo compongono e che ruoli ricoprano.

Abbiamo studiato la struttura tipica di un videogioco: quali sono le parti che lo compongono, quali sono le responsabilità di ciascuna di essa e come queste comunichino tra di loro. Si è visto come sia importante una corretta gestione degli input del giocatore attraverso la codifica in comandi.

Successivamente abbiamo visto come i game engines possano snellire il lavoro degli sviluppatori includendo al loro interno parte dell'architettura tipica dei videogiochi. Abbiamo individuato quali sono le soluzioni più valide e interessanti per il mondo degli smartphones e non solo. Paragonando le caratteristiche di diversi engines siamo riusciti poi ad individuare il framework migliore per le nostre esigenze. Siamo riusciti ad approfondire il funzionamento dell'engine Cocos2D grazie ai suoi sorgenti aperti.

Infine abbiamo affrontato concretamente le problematiche viste precedentemente realizzando un gioco per iPhone che sfruttasse alcuni dei sensori montati dal dispositivo. Utilizzando come input l'accelerometro e il touch screen si è riusciti a realizzare un'interfaccia di gioco semplice ed intuitiva. Testando l'applicazione su un dispositivo fisico abbiamo visto come questa scorra fluidamente con un consumo modesto di risorse.

Grazie all'uso del game engine Cocos2D siamo riusciti a velocizzare notevolmente lo sviluppo del gioco. Come abbiamo visto il game engine, insieme alle librerie messe a disposizione da iOS, racchiude in sé tutto lo strato dell'application layer del videogioco. L'engine non si limita a questo, definisce anche l'organizzazione dello scene graph e include una serie di classi che implementano parzialmente i nodi usati più frequentemente. Questo ha permesso di risparmiare una discreta fetta di progettazione e di implementazione.

L'uso di un game engine ha però portato con se anche qualche problematica. Il modo in cui è strutturato e in cui organizza i nodi non permette di separare completamente il lato di presentazione dallo stato del gioco e dalla sua parte computazionale. Cocos2D è infatti concepito per raccogliere nella struttura di un nodo sia lo stato di un'entità che la sua visualizzazione. Per cercare di risolvere questo problema si è realizzata la classe Entity che è composta da un nodo che contiene sia la parte computazionale che lo stato dell'entità, questa è legata alla sua rappresentazione grafica tramite un legame di aggregazione. Questa soluzione non ha comunque permesso di separare completamente lo stato di un'entità dalla sua parte computazionale.

Nell'ultimo anno si sono affacciati sul mercato nuovi smartphone e tablet dotati di processori dual core, come sviluppo futuro sarebbe sicuramente interessante approfondire in che modo si possa ripensare e riprogettare l'architettura di un gioco al fine di riuscire a sfruttare tutte le potenzialità offerte da questi nuovi dispositivi.

Ringraziamenti

Desidero ringraziare il professor Alessandro Ricci, relatore di questa tesi, per l'aiuto fornitomi durante la stesura.

Ringrazio con affetto i miei genitori, che, con il loro continuo sostegno morale ed economico mi hanno permesso di raggiungere questo traguardo.

La mia doppia famiglia merita un doppio ringraziamento per avermi fornito il doppio dell'affetto.

Ringrazio i miei nonni, i quali mi hanno sempre dimostrato un grandissimo affetto riempiendomi di attenzioni e nutrendomi sempre fino a farmi scoppiare.

Un ringraziamento speciale va a Lela che in tutti questi anni non ha mai finito di regalarmi bellissimi momenti. La ringrazio per avermi sopportato e supportato continuamente, con la sua immancabile allegria è sempre riuscita a strapparmi un sorriso o una risata, anche nei momenti più difficili.

Ringrazio infine i miei amici che mi hanno sempre offerto momenti di svago, in particolare Marans per le serate passate insieme a chiacchierare fino a tardi.

Bibliografia

[1] Ansca. Corona documentation. `http://developer.anscamobile.com/`.

[2] Apple. Apple ios development resources. `http://developer.apple.com/devcenter/ios/`.

[3] ARM. Arm architecture reference manual. `http://www.arm.com`.

[4] P. Bakhirev, P. Cabrera, I. Marsh, S. Penberthy, B. B. Smith, and E. Wing. *Beginning iPhone Games Development*. Apress, 2010.

[5] M. Carli. *Android Guida Per lo sviluppatore*. Apogeo, 2010.

[6] M. Carli. *Android 3 Guida Per lo sviluppatore*. Apogeo, 2011.

[7] Google. Android development resources. `http://developer.android.com/`.

[8] J. Gregory. *Game Engine Architecture*. A K Peters, Ltd., 2009.

[9] T. Isted. *Sviluppare applicazioni con Objective-C e Cocoa*. Apogeo, 2010.

[10] S. Itterheim and A. Löw. *Learn cocos2d Game Development with iOS 5*. Apress, 2011.

[11] M. McShaffry. *Game Coding Complete, Third Edition*. Course Technology Cengage Learning, 2009.

[12] Microsoft. Windows phone development resources. `http://msdn.microsoft.com/`.

[13] R. Quesada. Cocos2d-iphone documentation. `http://www.cocos2d-iphone.org/wiki/doku.php/`.

[14] S. Rabin. *Introduction to Game Development, Second Edition*. Course Technology Cengage Learning, 2009.

[15] Stonetrip. Shiva3d documetation. `http://www.stonetrip.com/`.

[16] K. Unger and J. Novak. *Game Development Essentials, Mobile Game Development*. Delmar Cengage Learning, 2011.

[17] Unity3D. Unity3d documentation. `http://unity3d.com/support/documentation/`.