ALMA MATER STUDIORUM
UNIVERSITÀ DEGLI STUDI DI BOLOGNA

Seconda Facoltà di Ingegneria
Corso di Laurea Magistrale in Ingegneria Informatica

SVILUPPO DI UN MULTIPLAYER ONLINE VIDEOGAME

Elaborata nel corso di: Fondamenti di computer graphics LM

Tesi di Laurea di:
CLAUDIO GIARDINI

Relatore:
Prof. SERENA MORIGI

ANNO ACCADEMICO 2011–2012
SESSIONE II

PAROLE CHIAVE

Grafica

Videogioco

Multipiattaforma

Programmazione concorrente

Sistema distribuito

Indice

Introduzione

Negli ultimi anni il mondo dei videogiochi ha conosciuto un profondo cambiamento, incoraggiato prevalentemente dalla recente evoluzione degli hardware disponibili. Negli anni tra il 2004 e il 2008, processori, memorie RAM e schede grafiche hanno conosciuto uno sviluppo incredibile delle loro capacità, sia in termini di velocità, che di funzionalità. Potendo contare su una rinnovata potenza e capacità di calcolo, l' industria videoludica si è enormemente rinnovata ed è riuscita ad espandersi notevolmente sul mercato con prodotti in grado di attirare l' attenzione grazie ad una qualità grafica in continuo miglioramento. Infatti è proprio in questo settore che si è maggiormente manifestato l' effetto di tale aumento di capacità e potenza di calcolo: se si guardano le immagini dei videogiochi precedenti a tale periodo, non solo si nota una profonda somiglianza nella grafica, ma saltano agli occhi oggetti spigolosi e grossolani che invadono una scena fatta di pochi colori sbiaditi e incapaci di far risaltare i pochi dettagli che era possibile inserire. Negli anni precedenti a questo periodo di evoluzione, a far da padrona sull' ancora ristretto mercato era la prima versione della consolle PlayStation. Poi, a partire delle prime schede video GeForce3, la fetta di mercato del videogame su pc è andata aumentando sempre più e con essa l' intero mercato videoludico. Sempre in quel periodo tra il 2004 e il 2008 un altro elemento rivoluzionante si è affermato con consapevolezza: il networking. I videogiochi, anche grazie ad un' altra crescita, quella della rete, si sono trasformati diventando dei multiplayer online e potendo così offrire nuove esperienze di gioco: in particolare nei generi "sparatutto" e "strategia in tempo reale" la possibilità di cooperare o affrontare altre persone non solo permetteva di colmare le enormi lacune dell' intelligenza artificiale, ma garantiva una certa longevità e novità. Il videogioco che più di

tutti ha segnato questo periodo è stato "World of Warcraft", che ha portato il multiplaying online al livello "massivo" (mmorpg: massive multiplayer online role playing game) e attorno al quale è sorta un' enorme comunità di appassionati. Quello che una ventina di anni fa era quasi un lusso – la disponibilità di un computer in casa, quasi esclusivamente per fini lavorativi – oggi è uno standard: tra le mura domestiche disponiamo di potenti pc che non sono più soltanto uno strumento lavorativo, sono anche un mezzo di informazione e anche uno strumento di intrattenimento per il tempo libero. Accanto a questi spesso abbiamo una consolle da gioco (non solo PlayStation, ma anche Wii, Xbox, ecc) e anche in questo caso ci sono stati enormi sviluppi, nelle stesse direzioni sopra citate. Accanto a questi due giganti del mercato videoludico, abbiamo assistito ad un' evoluzione anche per quanto riguarda i cellulari: non più solo semplici telefoni, ma oggetti sempre più sofisticati, che si connettono a internet, riproducono video e musica, fanno cose sempre più complesse, diventano "smart", e così in tempi molto recenti si è sviluppato il mobile gaming, il settore dei videogiochi per il piccolo smartphone, che è un settore sempre meno di nicchia.

Lo scopo di questa tesi è costruire un videogioco multiplayer online con l' obiettivo di giungere ad un' implementazione non solo per pc, ma anche per un ambiente diverso dal pc. L' applicazione costruita deve permettere il multiplaying non solo tra giocatori da pc, ma in generale tra tutti i giocatori che usano l' applicazione stessa, indipendentemente dall' ambiente di esecuzione (in questo senso si può dire che sarà multipiattaforma).

Più concretamente si fa riferimento a tre ambienti: il primo è quello del pc, per il quale è stata prodotta un' implementazione, il secondo è l' ambiente degli smartphone, e anche in questo caso è stata prodotta un' implementazione, e, per finire, il terzo ambiente è quello delle consolle di gioco, per il quale non è stata prodotta un' implementazione completa, ma sono stati effettuati degli esperimenti volti a verificarne la fattibilità.

Nel primo capitolo si intende prima fornire una panoramica degli aspetti funzionali di un videogioco (video, audio, intelligenza artificiale, networking, ecc) e poi analizzare i videogiochi multiplayer online dal punto di vista tecnico, riportando considerazioni su aspetti qua-

li la programmazione concorrente, i sistemi distribuiti e la computer grafica. Il secondo capitolo approfondisce la questione della computer grafica in maniera più specifica: si introducono le API (Application Programming Interface) OpenGL e OpenGLES, la pipeline fissa e programmabile. La parte finale di questo capitolo è dedicata agli aspetti di grafica utilizzati nell' applicazione sviluppata. Il terzo capitolo descrive il lavoro di analisi e progettazione svolto per questa tesi. Il quarto capitolo descrive la parte implementativa: riporta qualche breve considerazione relativa all' ambiente pc, illustra dettagliatamente lo sviluppo in Android, scelto come rappresentativo dell' ambiente smartphone, e descrive quella che è la situazione e le problematiche per quanto riguarda l' ambiente delle consolle di gioco. L' ultimo capitolo trae le conclusioni e delinea qualche possibile sviluppo futuro.

Capitolo 1

Caratteristiche di un videogame multiplayer online

1.1 Il videogioco percepito dall' utente

Quando si parla di videogame o videogioco due sono i concetti principali che vengono evocati: il concetto di video, che fa riferimento alla modalità di fruizione del gioco stesso, che è la visualizzazione a schermo, e poi che si tratta di un gioco, e ciò fa riferimento sia alle varie forme di interazione con l' utente che rappresentano gli input per il gioco, sia al modo in cui gli elementi del gioco evolvono nel tempo in base alle regole interne del gioco stesso e alla storia delle interazioni con l' utente. L' aspetto grafico e quello relativo ai comandi di input sono strettamente dipendenti dal sistema fisico sul quale il videogioco stesso è eseguito. Per ottenere un determinato livello grafico su schermo serve un certo supporto grafico da parte dell' hardware. E similmente le interazioni con l' utente variano a seconda dei diversi tipi di input messi a disposizione: dai cari vecchi mouse e tastiera, ai joypad, fino ai recenti touch-screen e sensori di movimento. L' aspetto relativo alla logica di gioco tende invece ad astrarre dal particolare hardware disponibile, anche se il modo in cui il gioco viene codificato dipende dal linguaggio utilizzato per programmare l' hardware. Questo vuol dire che il supporto fisico, almeno in parte, influenza la realizzazione

del gioco.

Oltre alla componente di gioco e a quella grafica, è possibile considerare anche una terza componente principale in un videogioco, che è quella sonora. Non è indispensabile, tuttavia arricchisce l' esperienza ludica, contribuendo talvolta ad evocare suggestioni e atmosfere che la componente video da sola non potrebbe fare.

Esistono poi altri possibili aspetti da considerare, più strettamente in relazione alla particolare tipologia di gioco che si intende realizzare, che sono stati via via aggiunti o considerati nel tempo. Il primo di questi aspetti minori è relativo all' intelligenza artificiale (IA), impiegata con successo nei primi videogiochi di logica (come dama o scacchi) riuscendo a garantire un buon livello di sfida al giocatore umano. Successivamente è stata usata per gestire il comportamento dei personaggi non giocanti nella maggior parte dei generi. Va comunque detto che con l' intelligenza artificiale vera e propria non ha nulla a che vedere: impiegare un vero processo di ragionamento virtuale è estremamente dispendioso in termini di potenza computazionale e non è certo adatto ad un contesto real time come quello dei videogiochi, si tratta più che altro di semplici comportamenti simulati. Un aspetto piuttosto recente è la gestione della fisica, simulare il comportamento degli oggetti virtuali come se fossero veri e propri oggetti reali, sottoposti a forze, deformazioni, esplosioni, ecc. La parte più complessa di questa componente riguarda la simulazione realistica di effetti come il fuoco, le esplosioni, oggetti che si rompono, vetri che cadono dalle finestre e vanno in frantumi, la fluidodinamica, ecc. Gli aspetti più semplici sono quelli relativi alla collision detection, per evitare la compenetrazione tra oggetti, attuato per lo più per mezzo di bounding boxes (parallelepipedi che contengono l' intero oggetto) Pur non costituendo dei veri e propri componenti, aspetti che si cerca di considerare nella realizzazione di un videogioco sono anche la trama (in certi generi) o la longevità (in quanto tempo si finisce il gioco) dal momento che questi fattori influenzano la desiderabilità da parte degli utenti.

Multiplayer fa riferimento alla possibilità di avere più giocatori reali nello stesso gioco, come suggerisce il termine stesso. Il multiplaying è un aspetto certamente non recente, la possibilità di avere più giocatori che interagiscono in modalità cooperativa o competitiva (o parzialmente cooperativa e parzialmente competitiva, a squadre)

tra loro e con il videogioco è sempre stato uno dei punti di forza delle consolle di gioco, nelle quali la prima forma di multiplaying è stata di tipo "single machine multi playing", ovvero più giocatori (fisicamente vicini) potevano interagire con lo stesso dispositivo grazie alla presenza di più controller. Con la sempre maggior diffusione delle linee digitali per le connessioni ad internet è stato possibile proporre anche delle funzionalità per permettere l' interazione tra giocatori fisicamente distanti, ognuno con il proprio dispositivo connesso con gli altri: si era diffuso il multiplaying online.

1.2 Panoramica degli aspetti tecnici di un videogame multiplayer online

La precedente sezione ha illustrato i videogiochi da un punto di vista "esterno" dell' utilizzatore, correlato alle funzionalità che l' utente percepisce del videogioco stesso. In questa sezione e nelle successive l' analisi verte su aspetti più tecnici e il punto di vista sarà quello del realizzatore. Se per un videogiocatore la percezione di un videogame è quella precedentemente esposta, per un (aspirante) ingegnere informatico che cos' è un videogioco e in particolare, dal momento che l' obiettivo di questa tesi è costruire un videogame multiplayer online, che cos' è un videogame multiplayer online? La prima e più ovvia risposta è che il videogame è un' applicazione della computer grafica e in effetti nessuno se la sentirebbe di contraddire tale affermazione. Poi, trattandosi di un multiplayer online, si ha a che fare con un sistema distribuito vero e proprio, ovvero un sistema software alcune delle cui parti possono risiedere e funzionare su macchine e piattaforme diverse ed eterogenee. Ad un livello di astrazione più basso è un' applicazione di rete, e ciò richiede anche buone conoscenze di programmazione di rete. Generalmente in un videogioco ci sono varie attività da gestire e portare avanti (la gestione dell' input dall' utente, la gestione delle dinamiche del gioco stesso, la resa su schermo, la gestione degli input dalla rete, ecc) per cui solide basi di programmazione concorrente sono spesso utili. E per finire, dal momento che molti aspetti e molte funzionalità di un videogioco sono fortemente platform-dependent (il tipo di dispositivo, il sistema operativo presente e le librerie suppor-

tate influiranno notevolmente) è bene decidere le piattaforme target per le quali sviluppare il prodotto e approfondire la conoscenza di queste. Tutte queste tematiche non sono singoli aspetti divisi e confinati, ma sono tra loro interconnessi, tuttavia, per rendere l' esposizione più semplice, verranno di seguito presentate una ad una, senza voler fornire una trattazione approfondita, puntuale ed esauriente, piuttosto di voler illustrare a grandi linee i concetti fondamentali utili alla comprensione del lavoro svolto.

1.3 Il videogioco come applicazione grafica

All' atto pratico scrivere un videogioco significa scrivere un software che produca un' animazione grafica su un qualche tipo di schermo. Questo implica ideare e strutturare una scena, collocandovi gli oggetti da visualizzare, gestire le luci e la telecamera. A complicare le cose c' è il fatto che il videogioco è una forma di applicazione real time, esistono dei vincoli temporali, anche se non rigidi, connessi l' uno al tempo di reazione e l' altro alla pazienza del giocatore umano: un videogioco esageratamente lento farà quasi certamente perdere la pazienza, se è troppo veloce sarà difficile da seguire e risulterà frustrante o ingiocabile. Sul piano della resa grafica questo vuol dire che esiste un certo frame rate da rispettare, ossia in un secondo deve essere visualizzato un certo numero di immagini in sequenza. Questo vuol dire che se l' hardware a disposizione ha una potenza limitata è necessario ottimizzare il software che disegna su schermo e la scena da disegnare. Nel primo caso si deve cercare di snellire il codice, nel secondo caso di limitare il numero di oggetti da disegnare e il dettaglio degli oggetti stessi. Esistono diverse tecniche per muoversi in questa direzione, come ad esempio l' uso di un livello di dettagli (Level Of Detail, LOD). Oggetti dettagliati ma molto lontani o molto piccoli possono essere rimpiazzati da oggetti simili ma molto meno dettagliati (questo significa usare dei modelli geometrici composti da molti poligoni in meno) con una perdita di qualità visiva molto limitata e un certo guadagno in velocità di esecuzione. Analogamente se la scena resa dovrà essere visualizzata sul piccolo schermo di uno smartphone sarà

necessario un dettaglio di gran lunga inferiore rispetto al caso in cui venga visualizzata sul monitor di un pc. La resa su schermo è poi un processo che deve essere svolto dal dispositivo, questo vuol dire che, anche se il gioco è un multiplayer online, la resa sul "proprio" schermo non può essere effettuata "altrove", ad esempio dal server di gioco e poi inviata, è un processo locale ai singoli dispositivi.

1.4 Aspetti di sistemi distribuiti

Come già accennato, un sistema distribuito è un sistema software costituito da parti software diverse ed eterogenee funzionanti su macchine e piattaforme (come sistemi operativi, middleware, ecc) distinte, diverse ed eterogenee. Perché una struttura del genere possa esistere si richiede alle varie parti la capacità di comunicare con le altre e un protocollo di comunicazione definito e condiviso perché le varie parti possano poi "comprendersi". Comune nei sistemi distribuiti è la caratteristica di "apertura": in qualunque momento una nuova parte può entrare nel sistema e in qualunque momento può uscirne. Ad esempio, nel caso del videogame che si vuole costruire, in qualunque momento un giocatore può unirsi agli altri e in qualunque momento può abbandonarli. Il sistema complessivo deve continuare a funzionare – e a funzionare correttamente – comunque. Legata a questa considerazione è la questione dei fault in un sistema distribuito: non deve accadere che un errore originatosi all' interno di una parte si propaghi in tutto il sistema abbattendolo completamente. Il fault deve essere parziale, se una parte del sistema va in crash il resto dovrà continuare a funzionare (più o meno) indisturbatamente. Un' altra questione molto importante è quella relativa alla sincronizzazione delle varie parti. A meno di non ricorrere ad una soluzione centralizzata, utile solo per sistemi non troppo grandi, sono necessari degli algoritmi per la sincronizzazione delle varie parti. In questo elaborato, trattandosi di un piccolo sistema, si è scelto di optare per la soluzione centralizzata, in cui una parte centrale svolge il ruolo di "server della sincronizzazione" di tutte le altre. Un altro aspetto da tenere in considerazione è il problema della coerenza e consistenza: tutte le parti devono avere la stessa visione dell' intero sistema. Nel concreto se un giocatore ef-

fettua un attacco e un altro cerca di schivarlo non deve succedere che per una parte l' attacco è andato a segno mentre per l' altra parte è andato a vuoto. Si deve determinare univocamente se l' attacco va a segno o va a vuoto e poi entrambe le parti dovranno "vedere" la stessa cosa.

1.5 Aspetti di programmazione di rete

Normalmente nel multiplaying online si sfrutta la rete per scambiare tutte le informazioni necessarie. Un' applicazione di rete è un' applicazione distribuita in cui almeno due parti sfruttano la rete per interfacciarsi l' un l' altra. La programmazione di rete è costruire applicazioni di questo tipo. L' architettura tipica di un' applicazione di rete prevede due ruoli diversi e ben distinti, il server e il client e stanno tra loro come il negoziante sta ai clienti. Il server è quella parte a cui i vari client si interfacciano per ottenere delle informazioni, delle elaborazioni e in generale un "servizio", la posizione del server è quella di colui che fornisce qualcosa, la posizione dei client è quella di coloro che chiedono. Dal punto di vista della dinamica delle interazioni, proprio come un negoziante che la mattina apre il suo negozio e attende l' arrivo dei clienti, il server si mette in "listening" e attende che i client remoti instaurino delle connessioni. Una volta connessi, client e server effettuano le operazioni necessarie (che dipendono dalla particolare applicazione) e si disconnettono. Nell' ultimo decennio si è diffusa un' architettura diversa, nota come peer-to-peer (p2p) , come i vari software di file sharing. Il realtà il p2p è un' architettura logica vista ad un livello di astrazione più alto, sotto al quale il funzionamento è sempre del tipo precedentemente descritto, non a caso il proprio client di eMule si connette al server (in realtà ad un server) di eMule e tutti i vari client connessi si vedono tra loro come pari. I vari giocatori che partecipano in multiplaying online ad un videogioco "si vedono tra loro" come pari. Ma ad un livello di astrazione inferiore sono tutti client del server di gioco.

1.6 Aspetti di programmazione concorrente

Per quanto non sia stato necessario fare ricorso a soluzioni particolarmente sofisticate, nel videogioco sviluppato per questa tesi esistono più attività da gestire contemporaneamente (gli input del giocatore, il gameplay, la comunicazione con il server, il rendering a schermo), le quali hanno tempistiche differenti e insistono sullo stesso insieme di dati. In generale un videogioco è un esempio di applicazione della programmazione concorrente. Quando si parla di programmare concorrente ci si riferisce alla costruzione di sistemi con più attività (come processi, thread) contemporaneamente in esecuzione, le quali attività possono avere qualche tipo di dipendenza l' un l' altra e di conseguenza possono tra loro interagire in qualche modo. Dal punto di vista dell' ingegneria, la concorrenza è utile a definire un adeguato livello di astrazione per quelle applicazioni che devono interagire con l' ambiente, eseguire più attività, gestire molteplici eventi, ecc. Inoltre permette di sfruttare meglio moderni hardware, ormai tutti quanti multicore. Ogni uso non banale della programmazione concorrente è basato sull' avere più processi che necessitano di interagire in un qualche modo (tra di loro per scambiarsi delle informazioni o per recuperare delle risorse esterne) , affinché il sistema complessivo possa raggiungere gli obiettivi per cui è stato costruito. Affinché il sistema complessivo funzioni è necessario sincronizzare le interazioni tra i vari processi affinché avvengano in un ben determinato modo (definire cioè un' ordine temporale tra le varie interazioni e far sì che venga rispettato) e imporre, quando necessarie, delle restrizioni nell' accesso a dati condivisi (mutua esclusione) per evitare delle corse critiche (race conditions, quando più processi accedono e aggiornano contemporaneamente risorse condivise e il risultato finale dipende dall' ordine con cui i due processi accedono). L' aspetto subdolo dell' avere contemporaneamente più di un flusso di esecuzione (più processi o più thread) è che le istruzioni scritte in un certo linguaggio di programmazione sono tradotte dal compilatore ciascuna in una piccola sequenza di istruzioni in linguaggio macchina. Per cui l' esecuzione di due istruzioni in linguaggio di alto livello di due processi differenti (una di un processo e una dell' altro) si concretizza nella sovrapposizione dell' esecuzione di due sequenze di istruzioni di

basso livello. Se vengono eseguite sulla stessa risorsa condivisa non c' è modo, per il programmatore, di sapere quale sarà l' ordine effettivo. Questo può causare aggiornamenti non corretti e perdite di informazioni. Per ovviare a tali problemi sono nate dapprima delle strutture dati per garantire l' esecuzione in mutua esclusione di porzioni di istruzioni di alto livello (semafori e monitor), poi alcuni linguaggi (come Java) hanno iniziato a gestire nativamente questa caratteristica. Gli stessi semafori già citati, assieme ad altre strutture dati più complesse come le barriere, si usano poi per gestire l' aspetto della sincronizzazione: vari legami temporali tra processi possono essere lo scambio di informazioni che deve avvenire in un certo ordine, oppure in un' esecuzione ciclica, prima di poter iniziare il ciclo successivo, tutti quanti i processi coinvolti devono aver completato ciascuno il proprio compito. Gestire la sincronizzazione dei processi è un aspetto di più alto livello rispetto a garantire l' esecuzione in mutua esclusione di porzioni di codice, e i vari linguaggi mettono a disposizione delle librerie con le strutture necessarie.

1.7 Il problema della scelta della piattaforma

La parola "piattaforma", nell' accezione con cui viene usata, necessita di ulteriori chiarimenti, in quanto può essere letta a differenti livelli. Ad un livello di lettura più grezzo e banale si tratta del tipo di dispositivo sul quale l' applicazione finale dovrà funzionare: nel caso esaminato in questa tesi vengono citati tre tipi di dispositivi, che sono pc, consolle di gioco e smartphone. (si rimanda al capitolo 4, quanto appena esposto è citato già in fase di definizione dei requisiti) "Piattaforma" non è da intendersi solo come tipo di dispositivo o come sistema operativo (ad esempio nel mondo pc i primi tre sistemi operativi sono Windows, Linux e Mac). Tutto ciò che si trova ad un livello di astrazione concettualmente inferiore rispetto a quello applicativo rientra nel concetto di piattaforma. Ad esempio le librerie grafiche: nel mondo pc un' abbondante maggioranza delle applicazioni per Windows fa uso di DirectX, mentre la parte rimanente usa OpenGL, anche questa supportata da Windows. Sui sistemi operativi non-windows, invece,

OpenGL è la libreria grafica di riferimento. Tra le varie consolle di gioco disponibili, sono fondamentalmente tre quelle che spiccano e si tratta di PlayStation3, Xbox360 e Nintendo Wii. In maniera analoga a Windows per quanto riguarda il settore pc, nel mondo consolle Xbox fa riferimento (per evidenti ragioni commerciali) a DirectX, mentre PS3 e Wii, pur supportando diverse librerie, hanno come libreria grafica di riferimento OpenGLES. Nel mondo degli smartphone esistono molte più possibilità di scelta (ci sono in circolazione numerosissimi modelli diversi), tuttavia le due principali famiglie sono iOS (iphone) e Android, dei quali, per scelte commerciali, il primo è un mondo chiuso e protetto, il secondo è un mondo completamente aperto e disponibile. Un eventuale sviluppatore desideroso di vendere la propria applicazione tramite il rispettivo app-store si trova nel primo caso di fronte ad un numero superiore di limitazioni e barriere rispetto al secondo caso. Per la grafica, nel settore degli smartphone, sebbene esistano vari standard supportati, OpenGLES è quello più usato.

Capitolo 2

Grafica per un videogame

Che si offrano esperienze grafiche simili al fotorealismo o che si presenti un mondo in una veste molto simile a quella dei cartoni animati, è ovvio che "l' occhio vuole la sua parte", come recita un noto modo di dire. L' aspetto grafico, in un videogioco, è sicuramente il più predominante, e lo dimostra il fatto le maggiori software house che occupano il mercato puntano sempre a rinnovare la qualità grafica dei propri prodotti, talvolta anche a scapito di altri elementi importanti, come la giocabilità, la longevità e a volte pure la fluidità.

Questo capitolo intende offrire una panoramica sulla computer grafica, disciplina che trova applicazione in tantissimi settori, quali la medicina, l' industria, il commercio, il settore militare, l' intrattenimento, avendo tuttavia come riferimento il mondo dei videogiochi, che rientra proprio in quest' ultimo settore.

2.1 Come in un film

Il modo migliore per iniziare la trattazione è il paragone con il mondo cinematografico e, in effetti, per la rappresentazione di una scena, la computer grafica usa gli stessi concetti che si ritrovano nella scena di un film: in entrambi i casi si posizionano degli oggetti, che vengono illuminati dalle luci e il tutto è ripreso da una telecamera. Ecco quindi i primi tre concetti:

- gli oggetti: la controparte degli oggetti fisici dei film è costituita da modelli tridimensionali definiti in un sistema di riferimento detto OCS, Object Coordinate System, proprio di ogni oggetto. Si tratta di rappresentazioni matematiche che possono essere di vario tipo, come mesh poligonali (una superficie di poligoni, di solito quadrati o triangoli) e superfici nurbs (funzioni che definiscono le proprietà geometriche dell' oggetto). Tali oggetti si portano dietro altri dati, come quelli relativi al colore (definito tramite colori solidi, o texture, o una combinazione di entrambi) e alle varie proprietà riflettenti e diffusive della superficie (i materiali).

- le luci: rappresentazioni matematiche di un insieme di proprietà legate, ad esempio, alle varie componenti luminose o al colore della luce stessa, possono essere di vario tipo, come la luce puntiforme, un punto luce ben definito che uniformemente irradia in ogni direzione la propria luce, oppure un riflettore, un cono di luce orientata in una ben precisa direzione.

- la telecamera: la controparte informatica della cinepresa, il modo più semplice per caratterizzarla è definirne la posizione, che corrisponde a dove si trova l' obiettivo, l' orientamento, aim, che corrisponde a dove è puntata, e l' up-vector, che indica la direzione verso l' alto. Ha anche altre caratteristiche, come la distanza focale e il campo visivo. Questi ultimi elementi permettono di introdurre un altro concetto fondamentale, quello del view-frustum, il campo di visione della camera, ovvero tutto ciò che la camera riesce a visualizzare dello spazio 3D ed in termini di grafica ciò che verrà visualizzato sullo schermo. Geometricamente il view frustum è una sorta di "piramide" formata da un piano immediatamente davanti alla camera (Near clip plane), uno più lontano (Far clip plane) e dai piani laterali che li uniscono.

Figura 2.1: Schematizzazione di una scena

2.2 Il processo grafico

Dopo aver introdotto gli ingredienti di base, la trattazione prosegue con la descrizione del processo grafico, per mezzo del quale viene prodotta l' immagine visualizzata sullo schermo. La figura 2.2 schematizza i macro-blocchi del processo grafico:

- la modellazione geometrica permette di ottenere gli oggetti (geometrici) descritti sopra, che popoleranno la scena finale. La modellazione può avvenire in diversi modi, quali:
  - per acquisizione, è possibile usare uno scanner 3D per catturare la forma di un oggetto reale
  - tramite modellazione procedurale, un algoritmo che fa uso di frattali e grammatiche di forme e costruisce il modello geometrico
  - sistemi particellari, insiemi di oggetti puntiformi soggetti a forze e vincoli, usati in grandi quantità possono produrre

nubi di polvere, fuoco, vetri in frantumi, vestiti, oggetti elastici, ecc

  - software di modellazione, che definiscono una funzione di modellazione iterativa (spline, nurbs) in base agli input dati via gui dall' utente, come lo spostamento dei punti di controllo o la modifica di altri parametri.

- animazione: mentre la modellazione è legata all' aspetto statico, l' animazione è legata all' aspetto dinamico, definire una relazione tra un' insieme di proprietà come posizione e angoli e ogni istante di tempo, e questo è possibile sia per i modelli geometrici, sia per le luci, sia per la telecamera.

- il rendering è la parte successiva a quella di modellazione e animazione. Obiettivo della fase di rendering è quello di produrre un' immagine bidimensionale a partire dalla scena tridimensionale e dalla telecamera. E' suddivisa in una prima fase di trasformazioni matematiche e una seconda fase di generazione di un' immagine a colori.

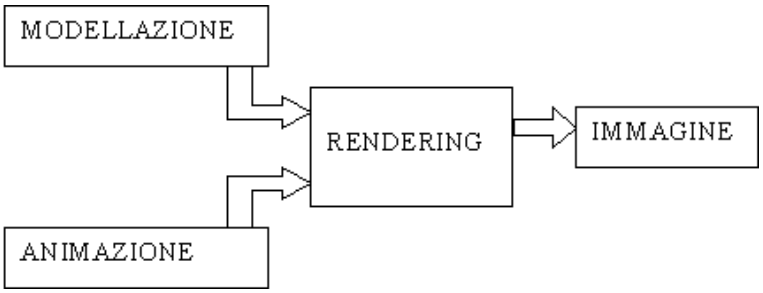

Figura 2.2: Schematizzazione del processo grafico

2.3 La fase di rendering

La fase più importante del processo grafico è quella di rendering. Esistono due principali modalità di rendering:

- pipeline based rendering (forward rendering): la scena è resa in una sequenza di passi che formano una pipeline. È composto da due stadi concettuali, il "geometry stage" e il "rasterization stage". In realtà a questi si antepone un "application stage", nel quale ogni oggetto geometrico è scomposto in un insieme di triangoli (tassellazione).

  - Il geometry stage avrà come input questi triangoli definiti nelle tre dimensioni. Durante il geometry stage si effettua una serie di operazioni sui vertici di tali triangoli:

    * avviene una prima trasformazione di coordinate: gli oggetti geometrici sono definiti in un sistema di riferimento locale, l' OCS. La prima trasformazione li colloca nel sistema di riferimento globale WCS, World Coordinate System, è la trasformazione di modeling. La seconda trasformazione serve a passare dal sistema di riferimento mondo al sistema camera, è la trasformazione di viewing.
    * avviene poi il calcolo dell' illuminazione dei vertici rispetto alle sorgenti luminose
    * se agli oggetti geometrici sono state associate delle texture, verranno generate le coordinate texture
    * avviene il clipping dell' immagine: tutto ciò che è al di fuori dal volume di vista viene rimosso. In questa fase avviene anche la trasformazione nel sistema di coordinate normalizzato: l' idea che si associa ad ogni vertice del view frustum il corrispondente vertice di un cubo definito in [-1,1] e tutto il resto è trasformato di conseguenza.
    * il sistema di coordinate normalizzato viene poi proiettato sul piano immagine, passando da un sistema di coordinate in tre dimensioni ad un sistema in due dimensioni (ICC, Image Coordinate System)
    * il passo finale è la trasformazione dall' immagine allo schermo, nello Screen Coordinate System

  - A questo punto si ha un insieme di triangoli in due dimensioni, e questi sono l' input del rasterization stage che non opera più sui vertici, ma sui pixel, anche in questo caso con una sequenza di operazioni:
    * sampling: ogni primitiva (triangolo) viene convertita in frammenti. Il frammento è una struttura dati che memorizza posizione, profondità, colore, coordinate texture. Si può dire che sia un pixel arricchito.
    * interpolazioni dei valori di luminosità, coordinate texture, profondità, ...
    * test di visibilità e rimozione delle superfici nascoste

- ray-tracing (backward rendering): una serie di raggi sono proiettati attraverso il piano di vista e quest'ultimo è colorato in base agli oggetti che i raggi intersecano. A partire da un punto sull' immagine si lavora quindi sulle primitive che sono state proiettate in questo. Mentre nel caso precedente l' hardware dedicato può supportare e accelerare il processo, in questo caso l' esecuzione è più a livello software. L' algoritmo è concettualmente più semplice: per ogni pixel si proietta un raggio che va dal centro di proiezione al volume di vista. Seguendo a ritroso il cammino del raggio si può calcolare la luminosità dell' oggetto, non solo rispetto alle fonti di luce, ma anche rispetto ad eventuali luci riflesse, è possibile gestire trasparenze e ombre e rimuovere tutto ciò che non è direttamente visibile.

2.4 Requisiti di un videogioco

L' elaborazione grafica può essere un processo leggero e veloce o lento e pesante, a seconda del risultato che si desidera ottenere. Nel caso delle produzioni cinematografiche si realizzano scene molto grandi, ad un livello di dettaglio altissimo, con una miriade di effetti speciali e con moltissimi oggetti in scena. Il film, per sua natura, può essere prodotto e visto in istanti diversi. Tra la produzione di un frame e la produzione del successivo può trascorrere molto tempo, e spesso gli studios impiegano grandi cloud di computer per accorciare i tempi.

Lo spettatore vede il film in un momento successivo alla resa delle scene e la questione del tempo necessario alle macchine per produrre un singolo frame non lo tocca, semplicemente riproduce una sequenza già registrata di immagini.

Non si può dire altrettanto per i videogiochi. In questo caso la resa della scena avviene davanti agli occhi del videogiocatore stesso. E viene effettuata dalla sua macchina, che di certo non è un sistema di computer in cloud. Inoltre la scena viene visualizzata su uno schermo che ha dimensioni molto inferiori e un dettaglio più grossolano. Inoltre, a differenza del film, che è un media passivo, nel videogioco è presente una componente di interattività. Tutto ciò per mostrare che nel videogame e nella grafica per videogame è presente una componente real time che non è presente invece nella grafica cinematografica. Mentre una factory di uno studio cinematografico può permettersi di rendere un frame anche in diversi minuti, la macchina (pc, consolle, smartphone) del videogiocatore deve renderla in poche frazioni di secondo.

Come già anticipato nel capitolo 2.1, il requisito real time è la presenza di un frame rate da rispettare, anche se non in senso stretto (soft real time), ossia è bene che in un secondo sia visualizzato un certo numero di immagini in sequenza. Questo ovviamente influisce sulla qualità dell' immagine finale. Si intuisce che il tempo necessario a rendere un frame è influenzato dal numero degli oggetti da rendere e in questo caso per oggetti si intende oggetti grafici elementari, come i triangoli e quadrati che compongono i modelli geometrici che popolano la scena. Per velocizzare la resa si deve ridurre il numero di "oggettini". Ma ovviamente questo significa avere un dettaglio più grossolano. Esistono varie tecniche per produrre in tempi ragionevoli una scena di qualità soddisfacente, di seguito se ne propongono alcune:

- la prima è stata già citata nel cap. 2.1, si tratta del L.O.D. e consiste nel disporre di diverse "risoluzioni" dello stesso modello geometrico (per l' esattezza di diversi modelli geometrici dello stesso oggetto, ciascuno realizzato con un numero maggiore o minore di poligoni) e rimpiazzare dinamicamente gli elementi lontani o quelli piccoli con le loro versioni a minor numero di poligoni

- utile è anche tener conto della curvatura di un oggetto: non ha senso usare migliaia di quadrati per disegnare una superficie piana, meglio concentrare la suddivisione in "faccette" nei punti a maggior curvatura, per evitare le spigolosità e usare poche grande "facce" per le superfici piane o poco curve (suddivisione adattativa)

- non disegnare cose che non si vedono: può sembrare banale, ma se nel mondo di gioco si è all' interno di un edificio, è inutile disegnare il mondo esterno: tutti quegli elementi verranno in ogni caso rimossi perché non visibili, ma richiederanno comunque del tempo di calcolo, che sarà stato sprecato, tanto vale, non caricare quei dati e non cercare di disegnarli

- usare le texture per aggiungere dettagli: questo è forse il metodo più usato, dettagli come cicatrici, rughe, vene gonfie, porosità della pelle su un volto, invece che ottenerli a partire da un modello geometrico che li abbia già di suo, quindi altamente rifinito e composto da tantissimi poligoni molto piccoli, è possibile ottenerli a partire da un modello geometrico molto più grezzo, composto da un numero di poligoni inferiore di uno, due o anche tre ordini di grandezza e "incollandoceli" sopra tramite l' applicazione di una texture sulla quale siano disegnati. Quello del volto è solo un esempio, è un metodo che funziona molto bene per tutta quella miriade di piccoli dettagli necessari a rendere "credibile" la scena: ciottoli nel terreno, sassolini, pavimentazione delle strade, fogliame, la pelliccia degli animali, decorazioni sulle automobili, ma anche su colonne e pareti, pieghe dei vestiti e, in generale, vale la regola empirica del "tutto ciò che è abbastanza piccolo da poter essere eliminato ma è necessario che non sia eliminato"

2.5 OpenGL

OpenGL (Open Graphics Library) è una specifica che definisce una API per più linguaggi e per più piattaforme per scrivere applicazioni che producono computer grafica 2D e 3D. L' interfaccia consiste in cir-

ca 250 diverse chiamate di funzione che si possono usare per disegnare complesse scene tridimensionali a partire da semplici primitive.

Come già detto, OpenGL è una specifica, ovvero si tratta semplicemente di un documento che descrive un insieme di funzioni ed il comportamento preciso che queste devono avere. Da questa specifica i produttori di hardware producono implementazioni, ovvero librerie di funzioni create rispettando quanto riportato sulla specifica OpenGL, facendo uso dell'accelerazione hardware ove possibile. I produttori devono comunque superare particolari test per certificare che i loro prodotti siano effettivamente implementazioni OpenGL.

La specifica di OpenGL è stata inizialmente supervisionata dall' OpenGL Architecture Review Board (ARB), formatosi nel 1992. L' ARB era composto da un gruppo di aziende interessate a creare un'API coerente e ampiamente disponibile. I membri fondatori dell'ARB comprendevano aziende del calibro di 3Dlabs, Apple Computer, ATI Technologies, Dell, IBM, Intel, NVIDIA, SGI, Sun Microsystems e Microsoft, che però ha abbandonato il gruppo nel marzo del 2003. Il coinvolgimento di così tante aziende con interessi molto diversificati, ha portato OpenGL a diventare nel tempo una API ad uso generico, con un ampio ventaglio di capacità. Il controllo di OpenGL è passato, ad inizio 2007, al consorzio Khronos Group, nel tentativo di migliorarne il marketing e di rimuovere le barriere tra lo sviluppo di OpenGL e OpenGL ES.

OpenGL assolve a due compiti fondamentali:

– nascondere la complessità di interfacciamento con acceleratori 3D differenti, offrendo al programmatore una API unica ed uniforme;

– nascondere le capacità offerte dai diversi acceleratori 3D, richiedendo che tutte le implementazioni supportino completamente l'insieme di funzioni OpenGL, ricorrendo ad un' emulazione software se necessario.

Il compito di OpenGL è quello di ricevere primitive come punti, linee e poligoni, e di convertirli in pixel (rasterizzazione). Ciò è realizzato attraverso una pipeline grafica (come precedentemente descritto). La maggior parte dei comandi OpenGL forniscono primitive alla pipeline grafica o istruiscono la pipeline su come elaborarle. Prima dell' introduzione di OpenGL 2.0, ogni stadio della pipeline realizzava una funzione fissa ed era configurabile solo entro certi limiti, ma dalla

versione 2.0 molti stadi sono totalmente programmabili attraverso il linguaggio GLSL.

OpenGL è una API procedurale che opera a basso livello, che richiede al programmatore i passi precisi per disegnare una scena. Questo approccio si pone in contrasto con le API descrittive ad alto livello le quali, operando su struttura dati ad albero (scene graph), richiedono al programmatore solo una descrizione generica della scena, occupandosi dei dettagli più complessi del rendering. La natura di OpenGL obbliga quindi i programmatori ad avere una buona conoscenza della pipeline grafica stessa, ma al contempo lascia una certa libertà per implementare complessi algoritmi di rendering.

Una delle caratteristiche più apprezzate in ambito professionale è la retrocompatibilità tra le diverse versioni di OpenGL: programmi scritti per la versione 1.0 della libreria devono funzionare senza modifiche su implementazioni che seguono la versione 2.1.

Al fine di imporre le sue caratteristiche multi-linguaggio e multi-piattaforma, vari binding e port sono stati sviluppati per OpenGL in molti linguaggi. Tra i più noti, la libreria Java 3D può appoggiarsi su OpenGL per sfruttare l' accelerazione hardware. Molto recentemente, Sun ha rilasciato le versioni beta del sistema JOGL, che fornisce binding ai comandi OpenGL in C, diversamente da Java 3D che non fornisce tale supporto a basso livello.

2.6 OpenGL ES

OpenGL ES è un sottoinsieme delle librerie grafiche OpenGL pensato per dispositivi integrati (come telefoni cellulari, ma anche per strumentazione scientifica e industriale). Viene gestito dal Gruppo Khronos, che cura anche lo sviluppo della libreria madre OpenGL.

Attualmente esistono alcune versioni diverse delle specifiche OpenGL ES. La versione 1.0 venne ricalcata sulla versione 1.3 di OpenGL, mentre la versione 1.1 si basa su OpenGL 1.5 e la 2.0 è definita in relazione a OpenGL 2.0. Sia la versione 1.0 che la 1.1 supportano due profili, common e common-lite; il profilo lite supporta solo tipi di dato interi invece dei normali dati in virgola mobile, mentre il profilo common li supporta entrambi. OpenGL ES prevede inoltre un altro

profilo safety-critical pensato per sistemi critici e/o ad alta sicurezza come display avionici, strumentazione per aeromobili e applicazioni militari: caratteristiche principali di questo profilo sono la stabilità, la testabilità e l'intrinseca robustezza.

- OpenGL ES 1.0 e 1.1: contiene molte funzionalità delle OpenGL originali e ne aggiunge alcune. Le due differenze principali sono la rimozione della semantica di chiamata glBegin ... glEnd per il rendering delle primitive (in favore dei vertex arrays) e l' introduzione dei dati in virgola fissa per le coordinate dei vertici delle primitive e degli attributi, per sfruttare meglio le capacità di calcolo dei processori embedded che spesso non hanno una FPU. Le altre differenze sono numerose ma di portata minore, quasi tutte semplificazioni per produrre un' interfaccia più semplice ed essenziale. OpenGL ES 1.0 è l' API 3D ufficiale sia in Symbian OS sia sulla piattaforma Android. Inoltre è supportata dalla PlayStation 3 come una delle API 3D ufficiali (l' altra è la libreria a basso livello libgcm). La PlayStation 3 include anche molte caratteristiche di OpenGL ES 2.0. OpenGL ES 1.1 è supportata da Android 1.6 e successive, oltre che da iPhone, iPod Touch e iPad

- OpenGL ES 2.0: rilasciata nel Marzo 2007, elimina la maggior parte della pipeline di rendering a funzioni fisse in favore di una programmabile: quasi tutte le funzioni di rendering della pipeline di transform e lighting, come il settaggio dei parametri di luci e materiali, vengono sostituite da vertex shaders e pixel shaders programmati separatamente. Per questo, OpenGL ES 2.0 non è retrocompatibile con le versioni 1.0 e 1.1. OpenGL ES 2.0 è supportata da iPhone (3GS e successivi) e iPad, supportata dalle più recenti versioni della piattaforma Android e scelta per WebGL (OpenGL per browser)

- OpenGL ES 3.0 Questa specifica è stata rilasciata pubblicamente nel mese di agosto 2012 ed è compatibile con OpenGL ES 2.0.

2.7 Grafica di PunchingBalls

Quest' ultima parte si concentra sulla descrizione degli aspetti grafici usati nel gioco. Nello specifico descrive la modellazione dei personaggi e degli altri elementi della scena, la loro resa e come avviene l' animazione.

2.7.1 Modellazione

I modelli geometrici usati in questo elaborato sono:
- un ring
- le facce degli 8 pugili
- il guantone da boxe
- il trofeo per il vincitore
- una stellina

Il totale è di 12 modelli. Di questi, gli ultimi 11 sono stati prodotti per modellazione geometrica, utilizzando come software di modellazione tridimensionale "Blender 2.49b", mentre il ring viene costruito attraverso una procedura. Parlare di modellazione procedurale sembra eccessivo, perché non si fa uso di grammatiche di forme, frattali, ecc, usati nella modellazione procedurale vera e propria, comunque si tratta di una funzione che costruisce il modello geometrico da sola, definendo i poligoni che costituiscono il ring e le loro posizioni in quello che sarà il modello finale del ring. Le dimensioni dei vari poligoni sono stabilite da dei parametri (quali il lato del ring, l' altezza della colonna) che permettono di modificare piuttosto facilmente alcuni aspetti della forma del modello finale (come lo spessore delle corde, l' altezza delle colonne, l' altezza del tappeto da terra, la posizione delle corde, ...) semplicemente cambiando questi pochi valori, come si può vedere dalla figura 2.3.

Per i motivi precedentemente spiegati (sezione 2.4), i modelli geometrici usati sono piuttosto semplici, ulteriori dettagli, come nel caso del ring l' intrecciatura delle corde, saranno poi aggiunti via texture, come dimostra la figura 2.4.

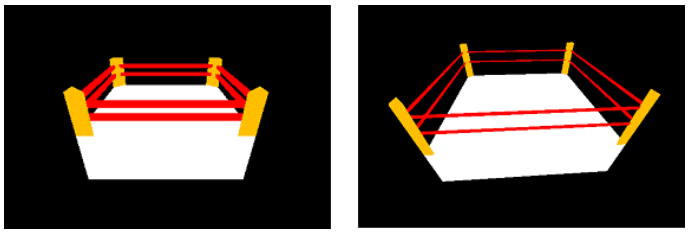

Figura 2.3: Il risultato della modellazione del ring, al variare dei diversi valori per i parametri delle misure

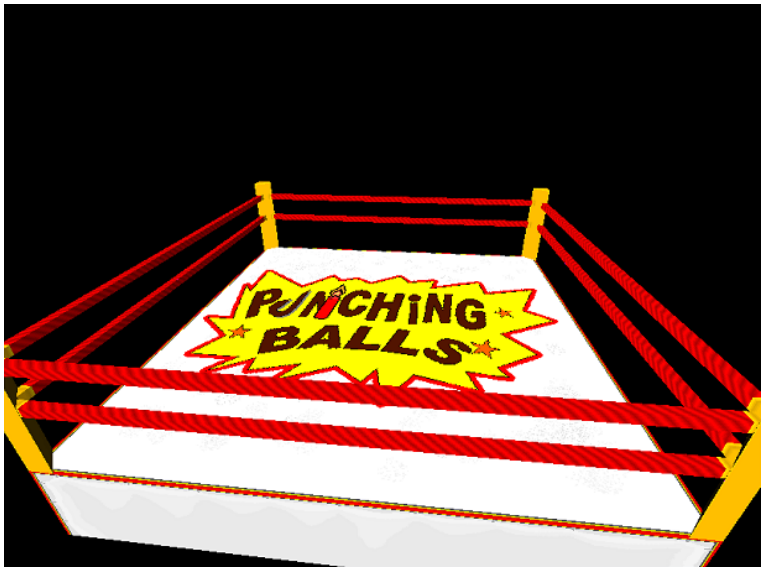

Figura 2.4: Aggiunta dei dettagli al ring via texture

Per quanto riguarda i modelli tridimensionali costruiti con il software di modellazione, anche questi sono piuttosto semplici (mediamente sui 2000 poligoni per le facce dei pugili, ridotti a 400 nella versione per smartphone, invece per gli oggetti più piccoli come la stella o il guantone da boxe il numero di vertici e quello delle facce scendono entrambi di un' ordine di grandezza).

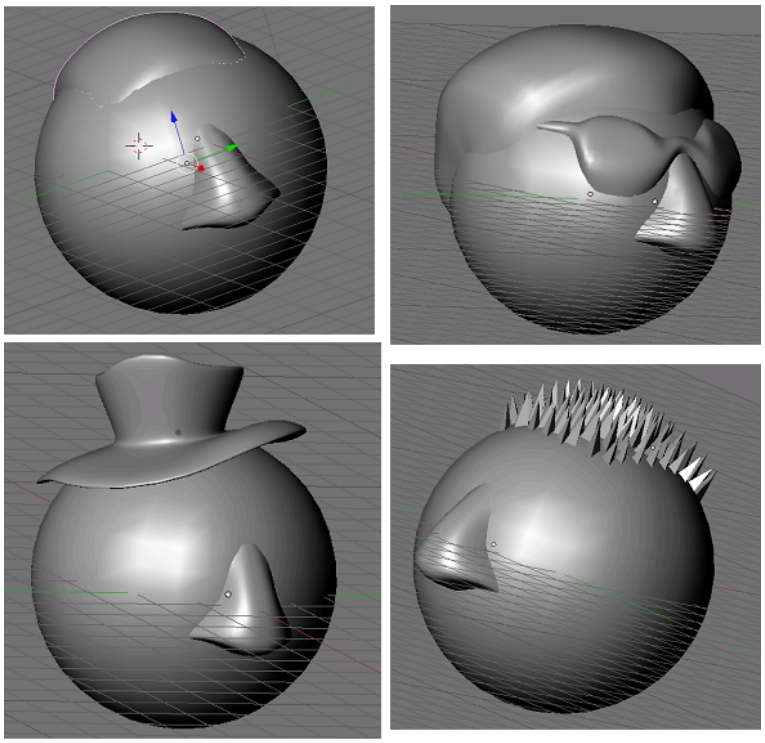

Figura 2.5: Una vista dei modelli geometrici delle facce dei personaggi

Semplicità dei modelli geometrici non vuol dire solo essere costituiti da pochi poligoni, ma soprattutto che ulteriori dettagli sono aggiunti via texture, come già più volte spiegato. Questo è vero soprattutto per le facce dei pugili, che fondamentalmente sono delle sfere con un naso attaccato come base di partenza per ciascun pugile, con una acconciatura dei capelli diversa per ognuno o qualche altro dettaglio, come un cappello o degli occhiali (figura 2.5). Ma a parte queste caratteristiche salienti, nei modelli geometrici non vi è altro, occhi, bocca, cicatrici,

collane e quanto altro saranno poi aggiunti via texture. I modelli poligonali dei pugili sono stati costruiti unendo insieme diverse superfici NURBS, di base la sfera e il naso per tutti, più altri elementi diversi a seconda del personaggio. Nella figura 2.6 sono mostrati alcuni istanti della nascita di uno dei personaggi.

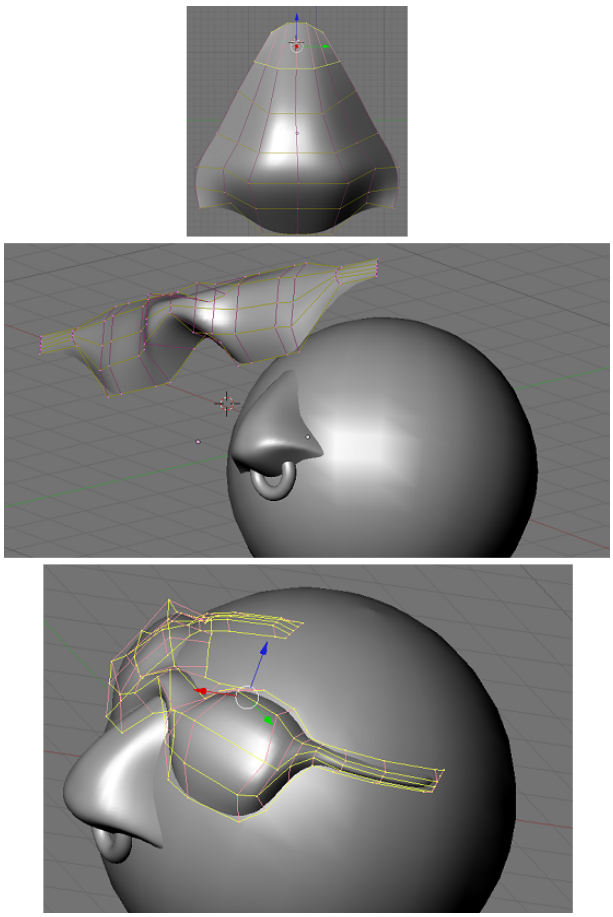

Figura 2.6: Istanti della modellazione geometrica della faccia di uno dei personaggi

Con ctrl + j Blender dà la possibilità di effettuare il join delle varie NURBS del modello. Ogni modello geometrico è stato poi esportato come mesh di poligoni nel formato Stanford disponibile dal menù Export (File: Export: Stanford(.ply)) di Blender. Questo permette di esportare delle mesh poligonali come lista di tutti i vertici, lista di tutte le facce (per ogni faccia la lista dei 4 vertici che la costituiscono) e lista delle normali ai vertici. Ai fini dell' applicazione interessava avere solo il modello senza preoccuparsi della texturizzazione, che sarebbe avvenuta in un secondo momento, tuttavia è stato interessante fare una prova già in questa fase usando la texturizzazione di Blender per avere un' anteprima di come sarebbe stato il personaggio finale, mostrata nella figura 2.7. Come già detto, il modello mostrato è costituito solo dalla sfera, il naso e le vene fuori dalla fronte, gli altri dettagli quali le cicatrici, gli occhi e la bocca sono stati "colorati" sopra al modello stesso tramite l' applicazione di una texture.

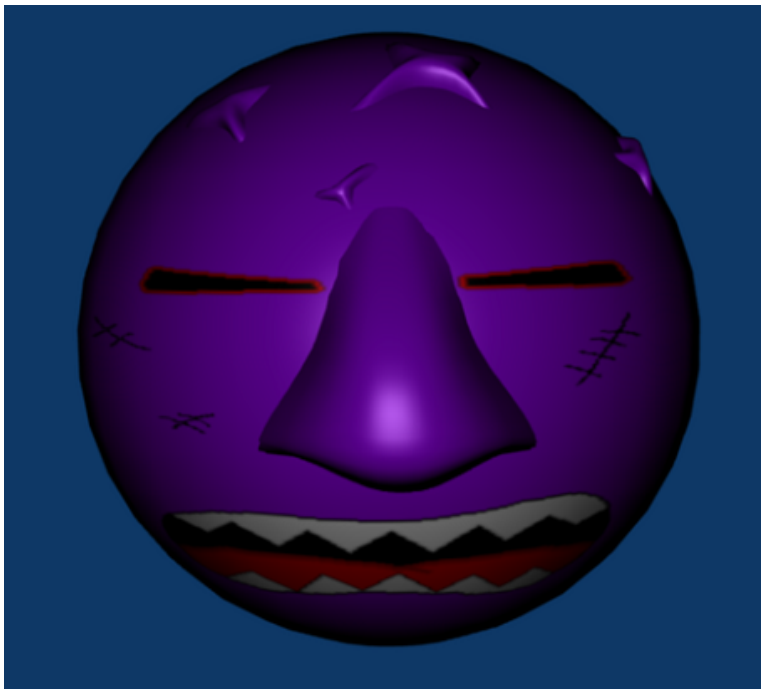

Figura 2.7: Anteprima in blender del risultato finale

Si propone di seguito un artwork raffigurante tutti i personaggi del gioco e mostra, per ognuno di questi, l' applicazione di quanto precedentemente illustrato. Risulta piuttosto semplice riuscire a distinguere quali parti sono state modellate e quali sono state aggiunte via texture.

Figura 2.8: Artwork raffigurante tutti i personaggi

Per costruire gli altri oggetti tridimensionali sono state esplorate altre tecniche di modellazione. In particolare il guantone e la stella sono stati ottenuti per skinning: come prima cosa si definiscono delle curve profilo ad altezze diverse, di dimensioni anche diverse, ma con lo stesso numero di punti di controllo (questo è fondamentale), poi le si uniscono tra loro (ctrl + j) e infine si esegue lo skinning vero e proprio con ctrl+f.

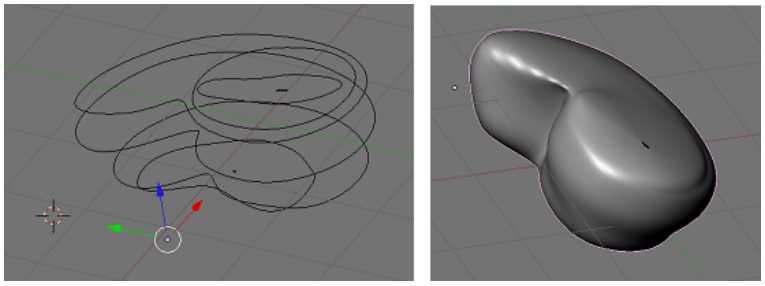

Figura 2.9: Modellazione del guantone da boxe tramite skinning

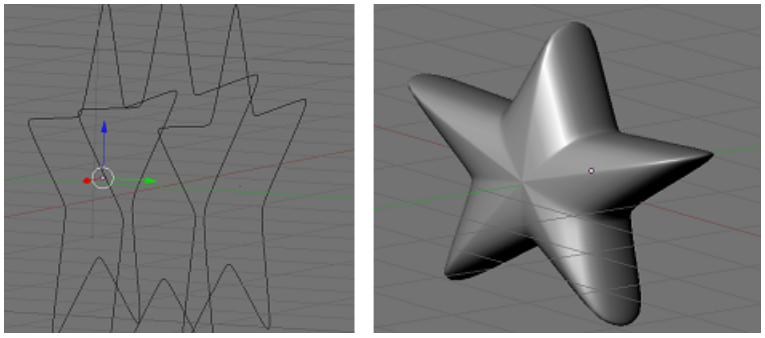

Figura 2.10: Modellazione tramite skinning della stella usata per l' effetto di stordimento

Per quanto riguarda il trofeo si tratta dell' unione di una base ottenuta da un cubo per estrusione, una coppa centrale ottenuta come solido di rotazione di una curva profilo e i due manici, uno la copia simmetrica dell' altro, ottenuti per sweeping di una piccola ellisse fatta

scorrere intorno alla curva più lunga che definisce la forma del manico stesso (l' ellisse ne definisce così lo spessore)

Figura 2.11: Risultato della modellazione geometrica del trofeo

2.7.2 Resa

Tre sono gli aspetti fondamentali da segnalare: luci, texture e ombre.

- Luci

  Trattandosi di un gioco che non punta al realismo, non è neppure necessario un rendering perfettamente fotorealistico che tenga conto di riflessi e contributi globali all' illuminazione. Il modello di illuminazione scelto è un modello locale, si tratta del gouraud shading, che funziona applicando il modello di illuminazione di phong ai vertici ed interpolando sulla superficie dei poligoni. Per quanto riguarda le normali ai vertici, il formato scelto per esportate i modelli poligonali (lo Stanford) contiene anche queste, e basterà solo caricarle dall' applicazione. Il materiale di base è un

qualcosa di simile alla porcellana. Questo consentirà di ottenere dei buoni highlight dei riflettori sulla testa dei pugili.

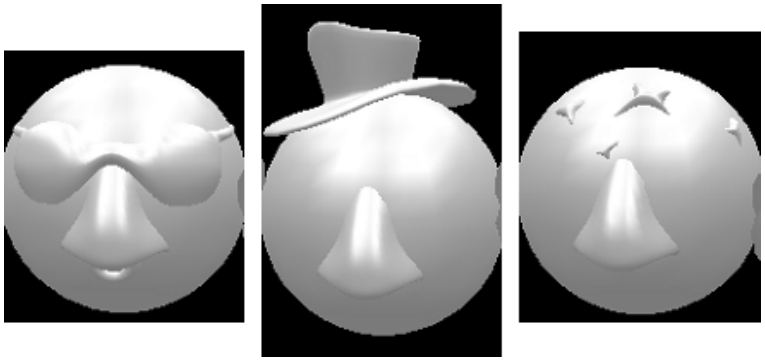

Figura 2.12: Applicazione del materiale ai pugili

I guantoni sono colorati solamente usando dei colori solidi e hanno un materiale di base più "gommoso" rispetto a quello della testa dei pugili. Ogni personaggio ha un proprio colore dei guantoni e questo lo si ottiene con un array di quattro coordinate per ogni pugile, che tiene traccia del colore di base dei guantoni indossati da quel pugile.

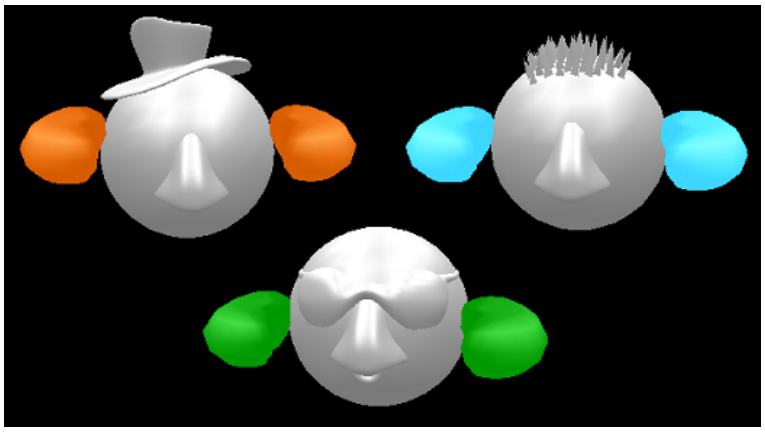

Figura 2.13: Applicazione del materiale anche ai guantoni da boxe

Per quello che riguarda la faccia il colore è ottenuto via texture e di seguito si spiegherà meglio tale aspetto. Per finire, una breve descrizione sulle luci usate: si tratta di 4 riflettori posizionati ai 4 angoli, con una distanza dal centro del ring doppia rispetto alla distanza tra il centro del ring e la colonna in quell' angolo e hanno un angolo di apertura (cutoff) di 30°. Oltre ai quattro riflettori esiste anche un contributo di illuminazione ambientale alla scena globale.

- Texture

  Le texture usate sono delle immagini di 512 per 512 pixel. In realtà nella versione per smartphone sono state notevolmente ridotte. Sono delle immagini in formato .raw o .png generate convertendo in tale formato delle bitmap disegante a mano libera. La scelta del formato .raw è dovuta al fatto che in tale formato si hanno solo dati puri, senza header iniziali e questo consente di leggerle molto facilmente con fread() su un buffer grande 512*512*3. L' interpretazione dei dati contenuti in questo buffer può essere direttamente effettuata con il comando glTexImage2D() senza ulteriori elaborazioni. Il formato .png è stato usato nella versione per smartphone, dal momento che Android mette a disposizione delle utilità per leggere tale formato con molta semplicità.

Figura 2.14: Alcune delle texture usate per la faccia

Via texture viene colorato tutto il ring, il trofeo per il vincitore e le facce dei pugili. Il mapping della texture al ring lo fa la stessa procedura che modella il ring, mentre il mapping delle texture sui

modelli delle facce dei pugili lo fa la funzione che legge i modelli geometrici. L' idea è quella di leggere i modelli geometrici e calcolare le coordinate texture subito, poi, in fase di disegno determinare la texture da usare a seconda di quale sia il pugile da disegnare, in modo da avere un' unica funzione di disegno a cui passare come parametri il modello 3d e la texture del pugile. Il mapping delle texture dei pugili sui rispettivi modelli geometrici è di tipo cilindrico. Questa scelta può sembrare contro-intuitiva per degli oggetti che sono poco più di una sfera, ma permette di disegnare le bitmap con relativa facilità, proporzionando la posizione dei dettagli nell' immagine con più accuratezza rispetto ad uno di tipo sferico.

- Ombre

  Le ombre presenti in scena sono degli oggetti aggiuntivi, in maniera analoga a come avviene nel gioco "Neverwinter Nights". È necessario fare così perché il modello di illuminazione scelto è locale e di suo non produce le ombre. Le ombre aggiunte alla scena sono le sole ombre dei pugili sul tappeto del ring e lo ombre delle corde e delle colonne sul tappeto (questo significa che le ombre di un pugile sull' altro o le ombre di un pugile su se stesso per le parti concave non sono presenti). Concretamente si tratta di disegnare poligoni posti ad una altezza lievemente superiore rispetto a quella del ring e colorati opportunamente con una texture che è simile in tutto e per tutto a quella del tappeto, tranne per il fatto che è scurita.

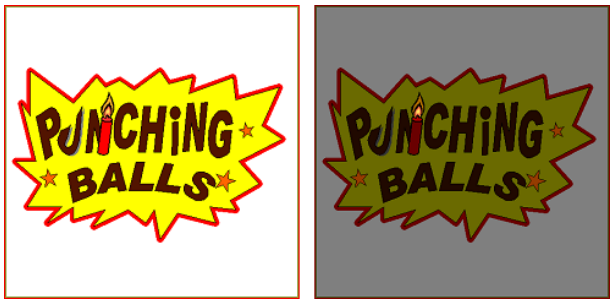

Figura 2.15: Confronto tra la texture del ring e quella usata per le ombre sul ring

Delle ombre considerate è possibile operare una suddivisione in due categorie: quelle statiche, dovute ad oggetti quali corde e colonne, che sono fissi ed è possibile definire una displaylist costante per queste e le ombre dinamiche, prodotte dai pugili e dalle loro mani, oggetti che non sono fissi ed è necessario calcolarle in tempo reale. La costruzione dell' ombra di un pugile rispetto ad uno dei riflettori non avviene in maniera esatta, ma approssimata, come mostrato nella figura 2.16: intanto la forma è approssimata, non si tiene conto di possibili irregolarità della sfera della faccia del pugile (irregolarità quali naso, cappello, acconciature e sporgenze varie) ma si disegna un' ombra circolare, un' altra approssimazione è dovuta al fatto che tale circonferenza è calcolata proiettando sul tappeto (a partire dalla sorgente luminosa) la circonferenza "equatoriale" della faccia del pugile, ovvero non si va a calcolare quali sono i punti esatti della sfera che sono tangenti ai raggi luminosi, un' ulteriore approssimazione è dovuta al fatto che in realtà non si tratta neppure di una circonferenza ma di un poligono regolare con molti vertici e come ultima approssimazione i guantoni da boxe producono un' ombra circolare (anche in questo caso un poligono a più vertici) e non con la forma del guantone stesso.

La scelta implementativa è stata quella di disegnare l' ombra di un pugile relativamente ad una sola sorgente luminosa, questo significa che è necessario usare quattro volte il disegno dell'

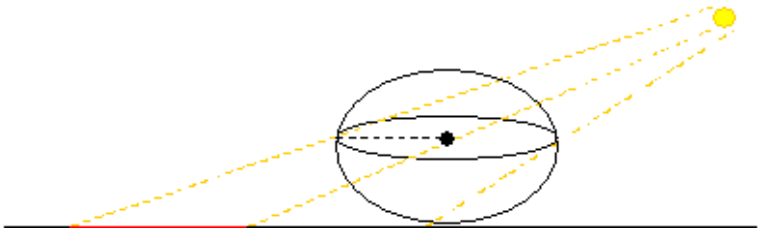

Figura 2.16: Schematizzazione di come avviene l' approssimazione per il calcolo delle ombre

ombra per ogni pugile e che non si ci si pone il problema delle zone "a maggior ombra", ovvero quelle zone in cui si intersecano più ombre e dovrebbero apparire più scure della semplice ombra singola. Nonostante tutto ciò, è stato possibile raggiungere lo stesso un risultato interessante, come si può vedere in figura 2.17.

2.7.3 Animazione e aspetti dinamici

I punti da segnalare sono i seguenti:

- Movimento

  Riguarda sia lo spostamento dei pugili sul ring, sia il movimento dei loro guantoni da boxe durante gli attacchi. Lo spostamento dei pugili sul ring è comandato pienamente dai giocatori, mentre il movimento dei guantoni è attivato dai giocatori premendo un pulsante di attacco, ma il percorso dei guantoni è calcolato dal computer. L' idea alla base dell' animazione dei guantoni è di cambiare un angolo di rotazione: il movimento scelto per i guantoni è piuttosto semplice, si tratta di un arco di circonferenza percorso prima in un senso e poi in quello opposto per riportare la mano in posizione iniziale, modellabile attraverso una rotazione rispetto al centro del pugile stesso (più o meno, in realtà il centro è "un po' più spostato in avanti"). L' animazione è calcolata utilizzando la definizione base di animazione stessa, cioè funzione di mapping tra variabili geometriche (posizioni e/o angoli, nel caso dei guantoni solo angoli) e istanti di tempo. Si tiene

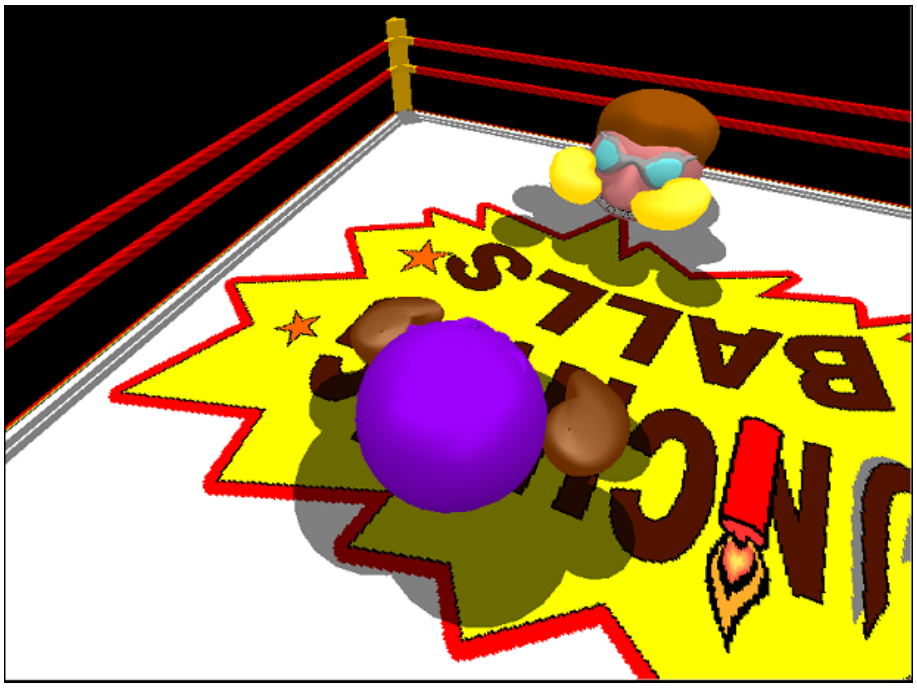

Figura 2.17: La scena finale arricchita con le ombre

traccia di quando l' attacco ha inizio e, in proporzione a quella che è la durata di un attacco si stabilisce l' angolo di rotazione dei guantoni. Il tutto funziona e produce l' animazione desiderata perché, in fase di disegno del frame, dopo aver disegnato il modello geometrico della testa del pugile, prima di disegnare un guantone, lo si ruota del corrispondente angolo indicato. Dal momento che questi angoli variano nel tempo il tutto produce l' effetto desiderato.

- Collision detection

  Si tratta di un aspetto legato al movimento, riguarda sia il fatto che i pugili, spostandosi, non devono uscire dal ring e non devono sovrapporsi, sia individuare i colpi andati a segno. Per le questioni riguardanti il movimento, si usa un semplice controllo sulle coordinate della posizione per non uscire dal ring e un

controllo un po' più complesso basato sulla distanza per evitare la compenetrazione tra i due pugili. Per rilevare i colpi andati a segno si fanno un po' di calcoli che coinvolgono le posizioni reciproche e quelle dei guantoni e, se il guantone dell' attaccante è troppo vicino all' attaccato, il colpo è considerato come andato a segno – a meno che l' attaccato non stia parando – e chi è attaccato perde dei punti salute in base al tipo di attacco subito. Esiste anche un attacco stordente che blocca per poco tempo chi l' ha subito, impedendogli di muoversi, di attaccare e di difendersi.

- Recupero della forza

  La forza è necessaria per poter sferrare gli attacchi e viene consumata in questo modo. Tuttavia, a differenza della salute (il danno causato da un attacco andato a segno si concretizza in una perdita di salute) la forza si rigenera col passare del tempo. Per fare ciò si tiene traccia di un valore che rappresenta l' intervallo di tempo dopo il quale si recupera un punto forza e, ogni volta che trascorre questo tempo, se la forza di un pugile è inferiore al massimo che potrebbe avere, viene aumentata di uno.

- Altri effetti

  Degli altri effetti animati, quello che coglie maggiormente l' attenzione è lo stordimento: un pugile stordito non potrà muoversi fino a che non si riprende e si vedranno alcune stelline girare sulla sua testa, ad indicare la condizione di stordimento, come mostrato nella figura 2.18. Anche in questo caso l' animazione è molto semplice, si sfrutta un angolo che cambia nel tempo. Effetti minori, ma realizzati nello stesso modo, sono la rotazione del trofeo per il vincitore e la rotazione del pugile su se stesso prima dell' inizio del match, nella fase di selezione del personaggio.

2.7.4 L' interfaccia utente

A completare la schermata ci saranno delle scritte a video, per riportare informazioni aggiuntive utili al giocatore. L' insieme di tutte

queste costituisce l' interfaccia per l' utente, utile anche a guidarlo quando potrebbe non sapere cosa fare. La figura 2.18 mostra anche l' interfaccia utente durante lo svolgimento di un match, mentre in figura 2.19 si mostra una schermata in cui l' interfaccia utente guida il giocatore, indicandogli come può agire. Per questioni legate alla dimensione dello schermo, nella versione per smartphone l' interfaccia utente da mostrare durante lo svolgimento del match è stata rimpiazzata dalle barre della salute e della forza, visualizzate direttamente sopra i personaggi (figura 2.20).

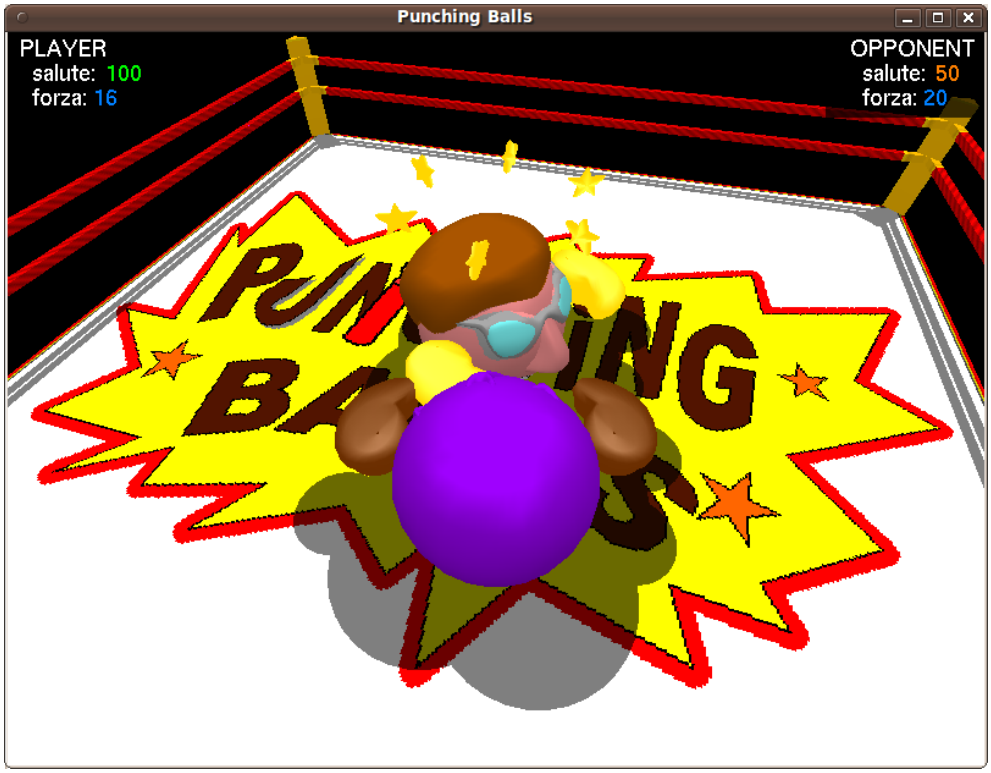

Figura 2.18: La scena finale arricchita con le ombre e l' interfaccia utente per mostrare ai giocatori l' andamento del match

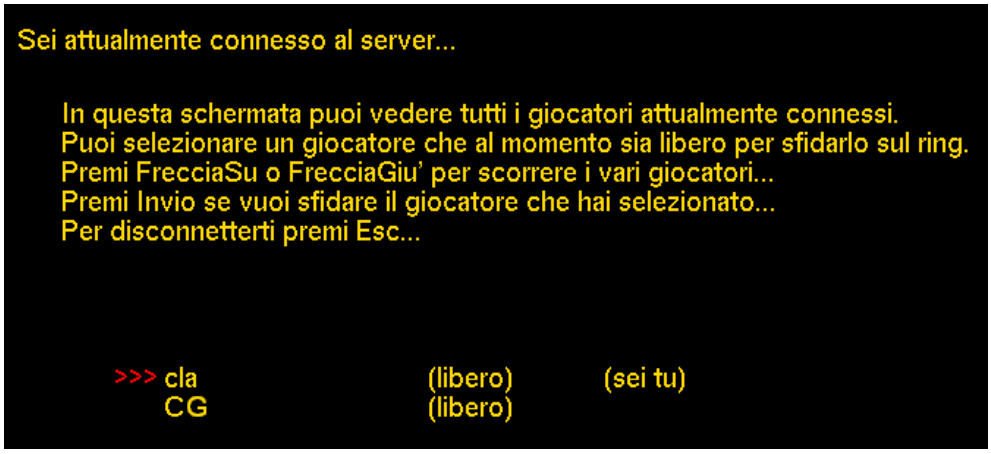

Figura 2.19: Un esempio di interfaccia utente

Figura 2.20: La versione per smartphone

Capitolo 3

Sviluppiamo il nostro gioco

3.1 Requisiti

Si intende realizzare un videogioco multiplayer online con un ulteriore requisito tecnologico, che è quello di rendere disponibile tale gioco non sono per pc, ma anche per piattaforme diverse, come lo smartphone e la consolle di gioco. Ciò significa che tale gioco deve consentire il multiplaying tra giocatori che usano il pc, giocatori che usano la consolle e giocatori che usano uno smartphone. In definitiva si tratta di un' applicazione in grado di mettere in comunicazione questi tre tipi di macchine.

Si vuole realizzare tale gioco dandogli una macro-architettura piuttosto simile a quella dei videogiochi professionali, quindi si vuole costruire un server di gioco con il quale i giocatori comunicano per mezzo dei propri client, che possono essere ciascuno in esecuzione su un pc, o su una consolle o su uno smartphone e in modo da dare ai giocatori la percezione di interagire direttamente con altri giocatori.

3.1.1 Descrizione del gioco

L' applicazione da realizzare è una versione esclusivamente multiplayer online del gioco "PunchingBalls", minigioco da me realizzato e ispirato al pugilato, in cui due personaggi giocanti (molto stilizzati, somiglianti a delle sfere con i guantoni da boxe) disputano un match di pugilato con l' obiettivo di colpire l' avversario per farne scendere la salute a

zero, mantenendo la salute del proprio personaggio al di sopra dello zero. Il primo che esaurisce tutta la propria salute ha perso.

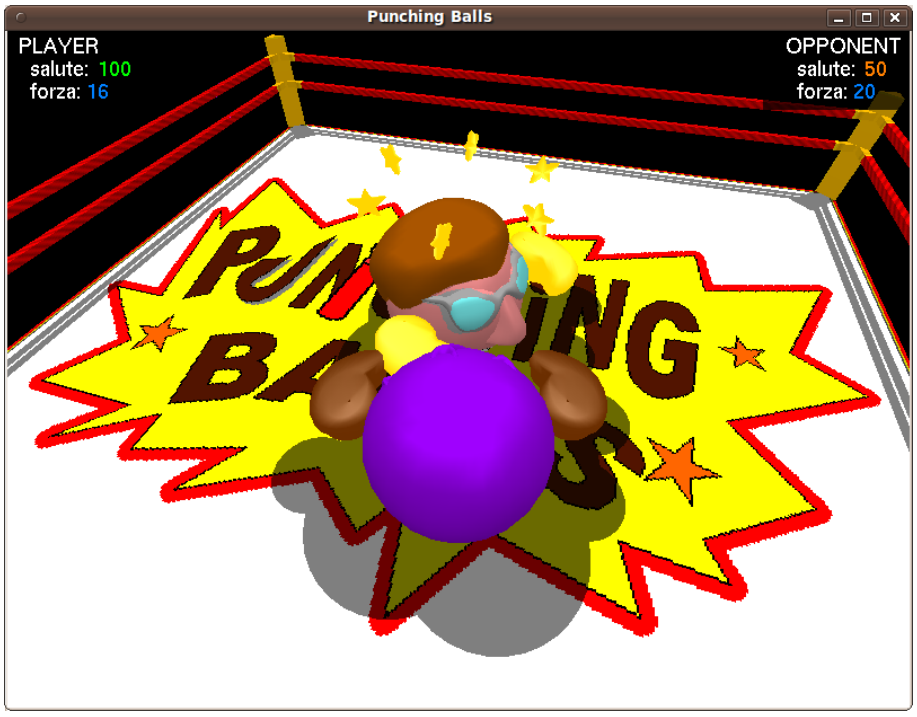

Figura 3.1: Una screenshot dalla prima versione di "PunchingBalls" pensata unicamente per il singleplayer

Nella versione singleplayer solo uno dei due era un personaggio giocante, mentre l' altro era controllato dal computer. Nella versione multiplayer, invece, ciascun giocatore dovrà controllare un proprio personaggio.

Come nei multiplayer online disponibili in commercio, dovrà essere presente un server di gioco che permetta di mettere insieme tutti i giocatori desiderosi di partecipare e che gestisca il gioco stesso.

Quando un giocatore avvia il proprio client, questo si connette al server di gioco, unendosi così a tutti gli altri giocatori. A connessione avvenuta si desidera che il giocatore possa visualizzare una schermata (anche piuttosto semplice) che fornisca tutta la lista dei partecipanti

Figura 3.2: Una schermata che mostra la lista dei partecipanti

(ovvero di tutti quei giocatori che sono al momento connessi), evidenziando chi di loro è "libero" e chi è "impegnato" (questo aspetto sarà spiegato tra poche righe) come mostrato nella figura 3.2.

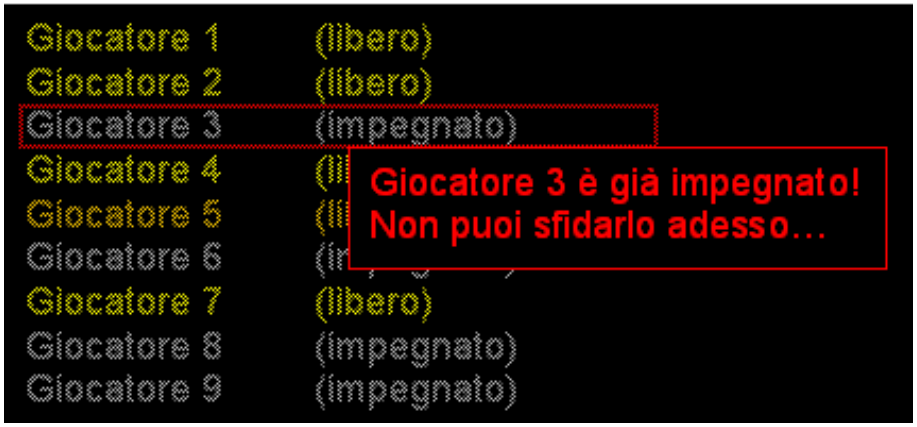

Figura 3.3: I giocatori impegnati non possono essere "disturbati"

A questo punto il giocatore che si è appena connesso è "libero" e può selezionare un altro dei giocatori liberi per sfidarlo ad un match. Se quest' ultimo accetta, sfidante e sfidato saranno ora "impegnati", nel senso che tra i due è in corso una sfida e pertanto nessun altro

giocatore potrà sfidarli se non prima che il match sia finito (come mostrato in figura 3.3).

Una volta che lo sfidato ha accettato la sfida del giocatore sfidante, i due giocatori avranno la possibilità di selezionare uno dei personaggi (pugili) con cui affrontarsi e avrà avvio il match vero e proprio, che termina quando uno dei due esaurisce tutta la salute. Quando il match finisce i due giocatori non sono più "impegnati", ma tornano "liberi".

3.2 Analisi dei requisiti

Si intende realizzare un' applicazione che consenta di mettere in comunicazione personal computer, consolle di gioco e smartphone. Dal punto di vista della distribuzione l' applicazione è divisa in un macro-blocco che costituisce la parte server e un altro macro-blocco che costituisce la parte client. Della parte client ne esisteranno diverse "copie", ciascuna in esecuzione su una piattaforma diversa. La parte server coordina e permette l' interazione delle varie parti client. Avendo questo ruolo, la parte server necessita di maggiori risorse di calcolo rispetto alla parte client e per questo si è deciso che dovrà risiedere su un pc. In definitiva si ha a che fare con un' applicazione distribuita in cui i vari client possono essere in esecuzione su piattaforme (hardware e software) diverse ed eterogenee ed interagiscono tra loro (ad esempio un client che risiede su un pc o una consolle di gioco interagisce con un altro client che risiede su smartphone) per mezzo del server.

Si consideri, per semplicità, un caso in cui si hanno due client che interagiscono con il server: il diagramma in figura 3.4 illustra come avviene l' interazione tra queste tre parti, come riportato nei requisiti.

Come riportato nei requisiti, ciascuno dei due client, al proprio avvio, si connette al server e riceve l' elenco di tutti i giocatori al momento connessi (partecipanti). Nel caso di A, essendo il primo, sarà anche l' unico a ricevere questa informazione. Tuttavia, alla successiva connessione di B, l' elenco aggiornato dei partecipanti viene inviato ad entrambi: questo serve ad informare B (che è appena entrato) su chi siano gli altri e ad informare A (che era già presente) del fatto che ora è presente un nuovo partecipante. Che si connetta prima A e poi B o viceversa è indifferente. Ad un certo punto uno dei due client (A)

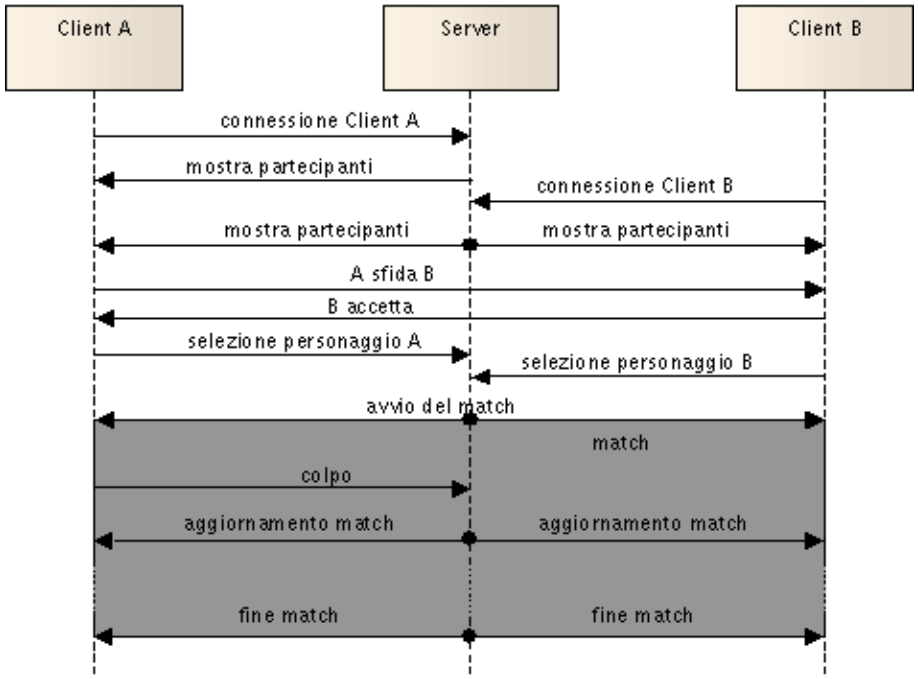

Figura 3.4: Diagramma delle interazioni che illustra lo scambio dei messaggi durante il gioco

decide di sfidare l' altro (B) e se lo sfidato accetta, dopo che entrambi avranno selezionato un pugile, può iniziare il match vero e proprio (rettangolo grigio) nel quale ogni client invia al server le proprie mosse e il server risponde aggiornando entrambi. Nella figura è indicato il caso in cui il client A sferri un colpo. Invierà un messaggio al server, che calcolerà se il colpo è andato a segno o meno e aggiornerà entrambi sulla nuova situazione, come mostrato. Non è necessario riportare tutto il match in figura, pertanto il resto è omesso. Comunque, a fine match, quando uno dei due giocatori avrà esaurito la salute, un messaggio di fine math sarà inviato ad entrambi per comunicare l' esito.

È possibile formalizzare il comportamento interno di entrambe le macroparti, client e server, mediante l' uso di diagrammi delle attività, come mostrato in figura 3.5 per il client e 3.6 per il server.

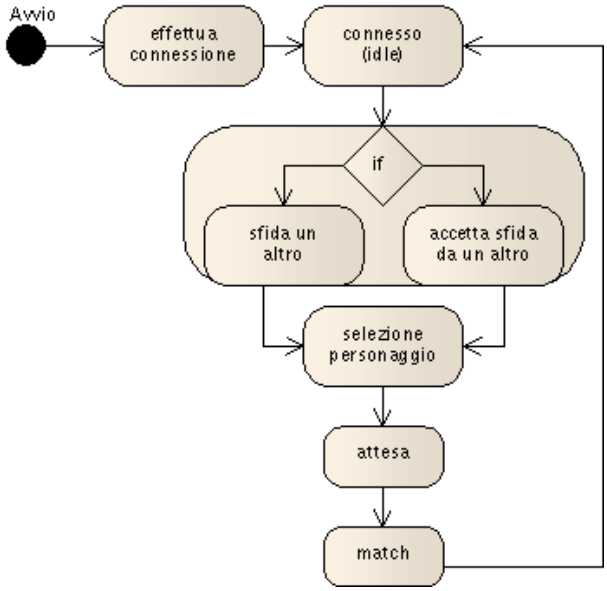

Figura 3.5: Diagramma delle attività del client

In base a quanto semplicemente riportato nei requisiti si ha che il client, all' avvio, effettua la connessione al server, una volta che è connesso si trova in una condizione di idle in cui può succedere o che riceva la sfida di un altro client o che esso stesso sfidi un altro client. In entrambi i casi si richiede di effettuare la selezione del personaggio, successivamente segue uno stato di attesa che precede lo stato del match. Quando il match termina torna nello stato di idle.

Un po' più complicata è la situazione della parte server, mostrata nella figura 3.7, in quanto non è facilmente rappresentabile con un unico diagramma sequenziale. Fondamentalmente ha due aspetti da gestire: il primo è l' accettazione di nuovi client, il secondo è la gestione dei vari match. Per ognuno di tali aspetti si riporta un diagramma. I due aspetti devono essere gestiti parallelamente e a volte possono interagire tra loro.

È possibile comporre i grafici fin qui riportati (figure 3.4, 3.5 e 3.6) per dare una visione della dinamica complessiva, avendo ancora come riferimento il caso già esaminato di un server e due client in esecuzione,

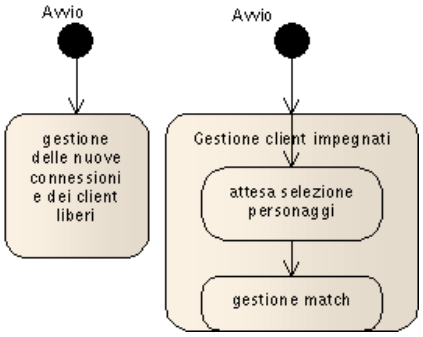

Figura 3.6: Diagramma delle attività del server

così da mostrare come siano tra loro correlati il comportamento interno di client e server appena illustrati e l' aspetto dell' interazione a livello di applicazione distribuita.

Il grafico riportato in figura 3.7 combina i singoli grafici degli stati con il grafico delle interazioni tra client e server. Anche se formalmente non esiste un diagramma del genere (gli strumenti classici come gli interaction-diagram e le statechart riescono a catturare solo una prospettiva del sistema complessivo o una sua parte) il grafico 3.7 è stato costruito proprio per le finalità sopra riportate. Per distinguere i vari aspetti, le frecce simboleggianti dei messaggi sono state colorate in rosso, mentre, solo per i client, in azzurro si riporta il flusso del comportamento interno effettivamente seguito, così come spiegato sopra (in grigio, invece, i rami non percorsi).

Infine, dal punto di vista funzionale, è richiesto che le due macroparti client e server, oltre ad effettuare delle elaborazioni di dati e calcoli, abbiano la capacità di eseguire certi tipi di operazioni che sono:

- per il client:
  - comunicare con il server (inviare e ricevere messaggi)
  - interagire con l' utente umano, il che significa:
    - gestire degli input
    - disegnare sullo schermo

    - riprodurre dell' audio

- per il server:
    - comunicare con i client

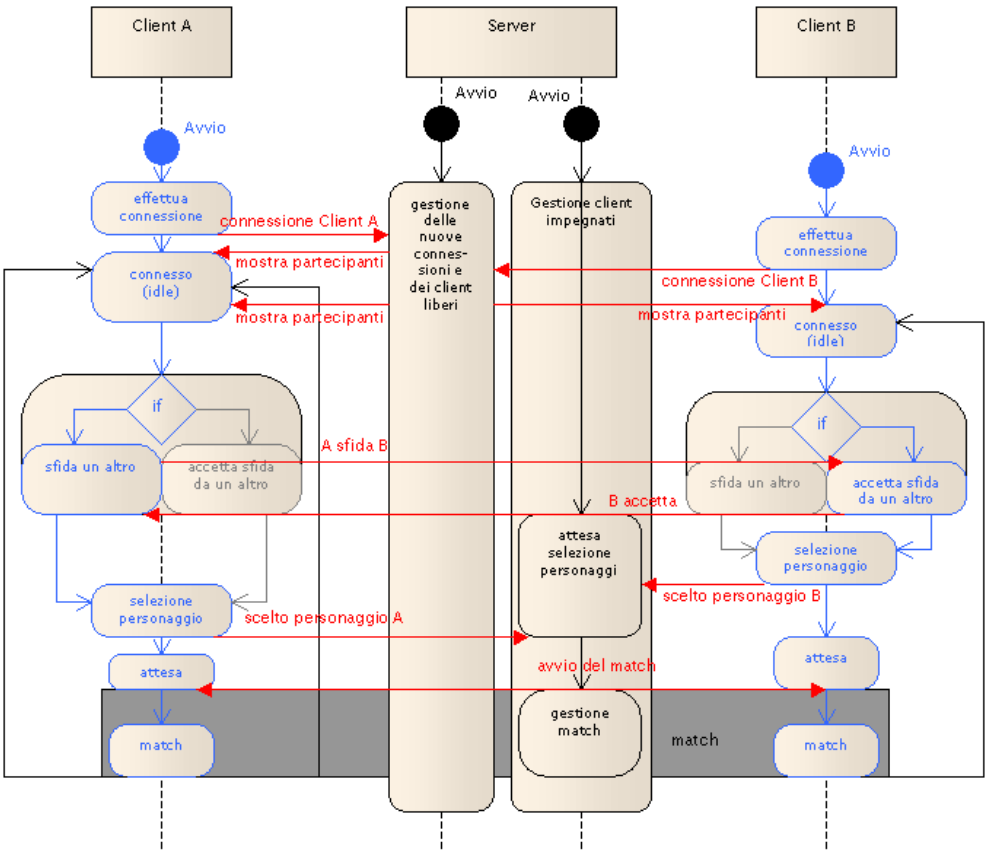

Figura 3.7: Visione complessiva dei comportamenti interni e delle interazioni tra due client e un server

3.3 Progettazione

Dalla precedente analisi si deduce che gli aspetti relativi all' interazione e al comportamento sono stati già abbastanza ben delineati nei

requisiti, di conseguenza la progettazione di questi due aspetti ricalcherà molto quanto precedentemente riportato e lo approfondirà ove necessario. L' aspetto strutturale è invece a malapena abbozzato, in quanto si è fissato solo che dal punto di vista architetturale si avrà una macroparte server e una macroparte client. In fase di progettazione, solitamente, la parte strutturale è quella con il maggior grado di libertà e in questo caso particolare ci si riferisce alla struttura interna di client e server, che non è stata minimamente esplicitata e che va quindi sviluppata del tutto. Tra le varie possibilità a disposizione si è deciso di sfruttare come valido suggerimento l' analisi funzionale e di strutturare client e server su base funzionale. Si decide (come mostrato in figura 3.8) di strutturare quindi il client in tre grandi sezioni, una dedicata ad ospitare la "client business logic" (Core), una dedicata a gestire l' aspetto comunicativo (Network) e la terza dedicata alla gestione di input e output da e verso l' utente (UserInterfacing). Quest' ultima viene ulteriormente scomposta in una sottosezione dedicata a gestire gli input come le pressioni dei tasti (InputHandling) , un' altra per disegnare su schermo (Graphics Kernel) e una terza per gestire l' audio (Audio).

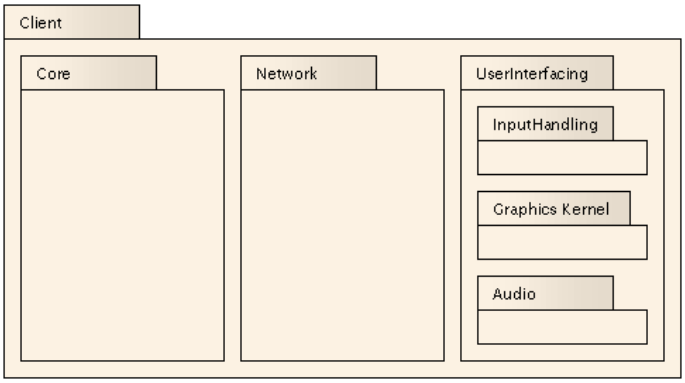

Figura 3.8: Diagramma strutturale del client

Si è deciso inoltre di riflettere la struttura architetturale anche a livello di "entità funzionanti" (si vogliano chiamare thread, processi, o più genericamente flussi di controllo, mostrati in figura 3.9).

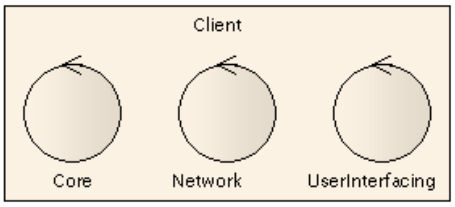

Figura 3.9: Rappresentazione delle entità computazionali del client

Strutturalmente più semplice è la parte server (figura 3.10), che si suddivide in una (macro)sezione dedicata alla "server business logic" (Core – che è diverso dal Core del client) e l' altra per gestire l' aspetto comunicativo (Network).

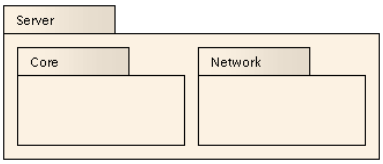

Figura 3.10: Diagramma strutturale del server

È possibile raffinare e approfondire meglio l' architettura fin qui presentata.

Cominciando dal client, a cui fa riferimento la figura 3.11, nella sottosezione "Core" (del Client) si trova ora un' ulteriore sottosezione, la "IdleLoop": dal momento che si è optato di gestire la parte core con un flusso di controllo dedicato, la sezione "IdleLoop" dovrà contenere la definizione di questo flusso di controllo. In maniera analoga "Network" è stata strutturata per poter gestire separatamente i messaggi che entrano e quelli che escono e il processo che si prende carico di gestire l' aspetto comunicativo dovrà occuparsi solo della parte di ricezione, dunque è definito dentro "In".

La sezione che è stata dettagliata di più è quella relativa alla gestione dell' utente, già precedentemente suddivisa in "InputHandling" e "Graphics". La prima ha una sottosezione dedicata alla configurazione degli input, una per la gestione degli input da tastiera e una terza per

la gestione degli input via mouse. La sezione per il disegno dello schermo ("Graphics") è stata anche questa divisa in tre sotto-sezioni, una

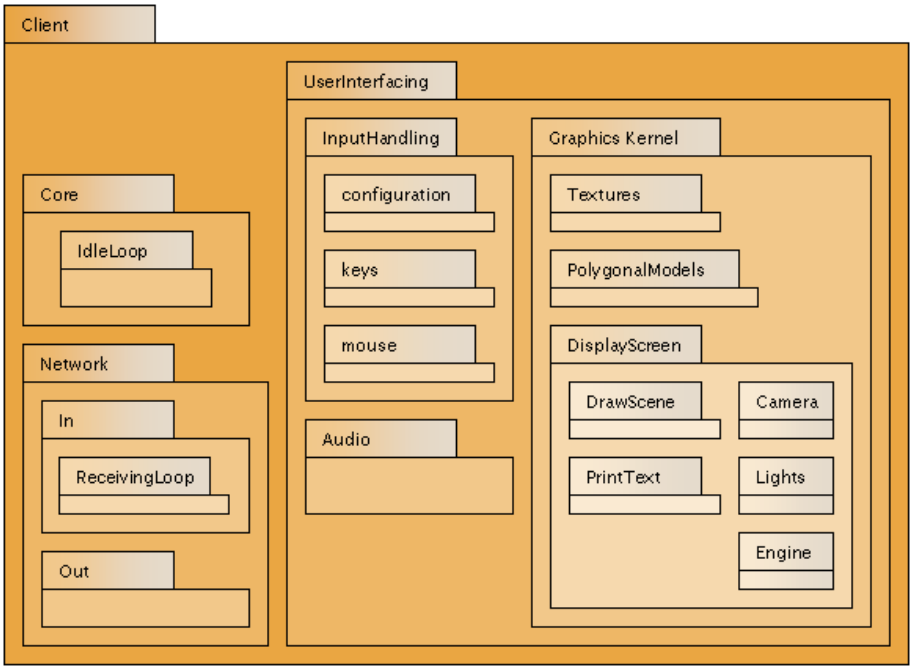

Figura 3.11: Diagramma strutturale del client, maggiormente dettagliato

per gestire le texture, una per gestire i modelli 3d e una terza per effettuare il disegno a schermo vero e proprio ("DisplayScreen"). In quest' ultima gli aspetti da curare sono camera ("Camera"), luci ("Lights"), oggetti in scena ("DrawScene") e scritte a schermo ("PrintText") e tutto ciò dovrà "andare in pasto" all' Engine grafico.

Più semplice è il dettaglio per il Server, riportato nella figura 3.12, in cui si possono riconoscere le relative controparti già descritte per il Client.

Vale la pena di fare un' osservazione non banale: si noti che, nonostante il client abbia a che fare con aspetti molto platform-dependent (invio e ricezione di messaggi in rete, gestione di input, grafica, ...) l' architettura di progetto fin qui realizzata è – o cerca di essere il più possibile – platform-independent, il che significa che può essere

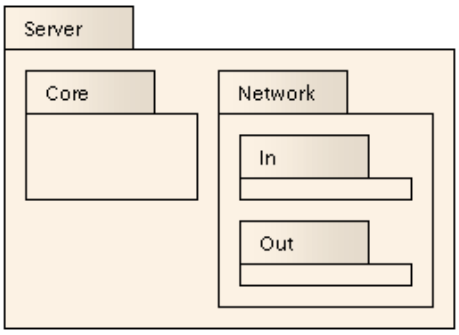

Figura 3.12: Diagramma strutturale del server, maggiormente dettagliato

implementata – eventualmente con l' aggiunta di ulteriori particolari – su un sistema operativo Windows così come su un Linux o un Mac, può essere scritta con il linguaggio o i linguaggi di programmazione che si preferisce (o che risultano più efficienti e/o efficaci) e la parte di grafica può essere realizzata con il software che si preferisce (che sia la libreria OpenGL o DirectX)

L' aspetto comportamentale relativo a client e server rimane fedele a quanto già delineato precedentemente, con qualche piccola aggiunta per riempire alcuni dettagli precedentemente tralasciati.

Cominciando dal client, raffigurato in figura 3.13, non è detto che la connessione al server vada sempre a buon fine. In particolare possono accadere due scenari alternativi: il primo è il caso di un errore di sistema (non funziona la rete, indirizzo del server errato, il server non è funzionante, ecc) e il secondo è quello di un rifiuto esplicito da parte del server: si è detto che il server permette ai client di riunirsi, li raduna, ma se ne accetta troppi c' è il rischio che si degradino le prestazioni. Quindi si fissa una soglia, un numero massimo di N client possono essere contemporaneamente connessi al server e se un client N+1 prova a connettersi verrà respinto.

Sono stati aggiunti anche altri stati. Uno di questi serve a dare al giocatore la possibilità di scegliersi un proprio "nickname" , mentre lo stato successivo permette di specificare il server al quale connettersi.

Successivamente allo stato di idle si incontra un gruppo di due

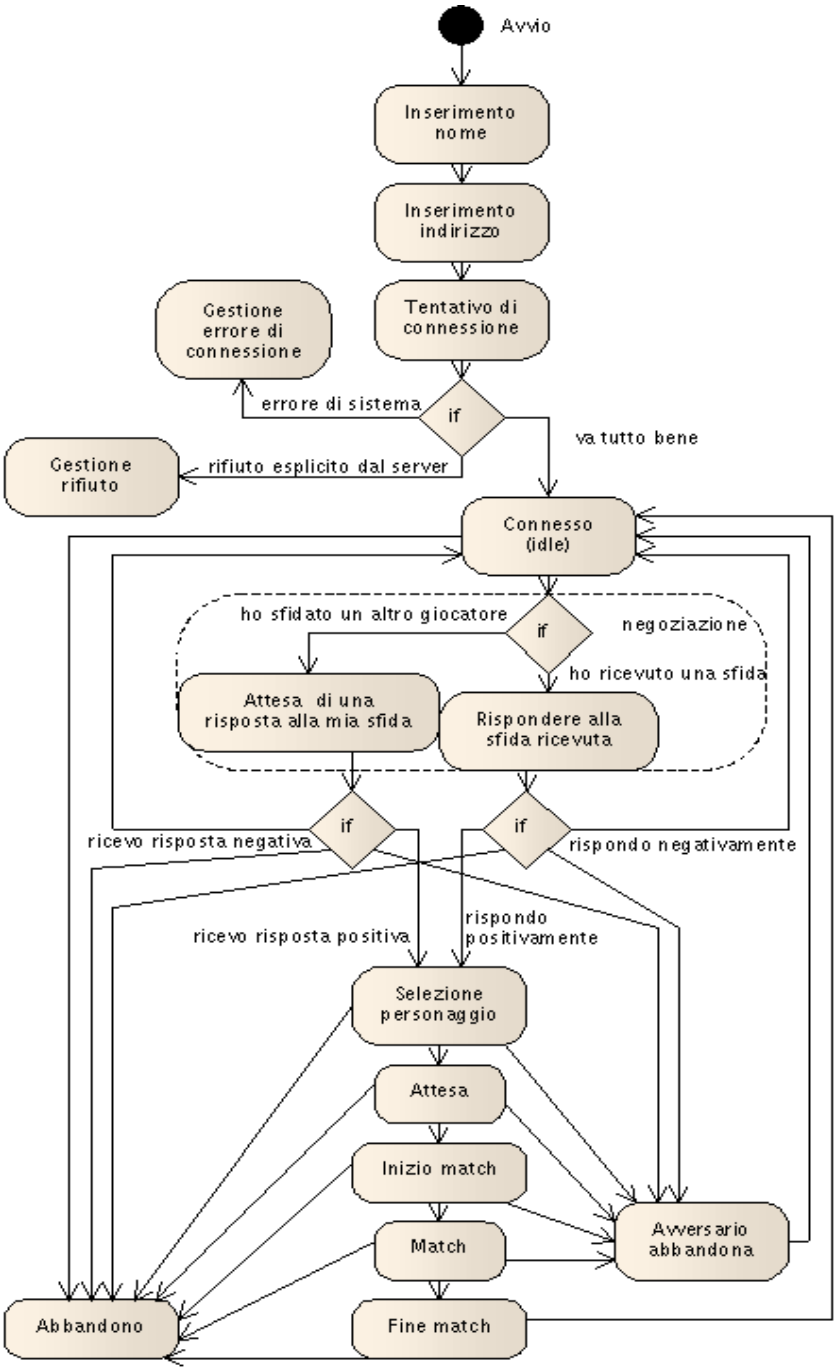

Figura 3.13: Diagramma delle attività che illustra con maggiore dettaglio il comportamento del client

stati aggregati nell' ovale chiamato "Negoziazione" e si tratta di due percorsi paralleli e duali l' uno rispetto all' altro, tramite i quali due giocatori possono mettersi d' accordo sul disputare un match. Se, ad esempio, un giocatore decide di sfidarne un altro, uscirà dallo stato di idle inviando una richiesta all' altro giocatore e dovrà attenderne la risposta. L' altro giocatore, invece, uscirà dallo stato di idle ricevendo tale richiesta e uscirà dallo stato di sfidato rispondendo a tale richiesta.

Quando entrambi (sfidante e sfidato) concordano per voler disputare un match lo stato di selezione del personaggio dà la possibilità di scegliere uno dei personaggi disponibili con cui affrontare la sfida. Una volta che il giocatore ha selezionato il proprio personaggio, invierà tale informazione al server. Dal momento che è praticamente impossibile che i due giocatori effettuino la selezione del personaggio esattamente nello stesso istante, il successivo stato di attesa serve a tamponare il ritardo di uno dei due (concettualmente significa che A ha già scelto e sta aspettando che B si decida o viceversa). Lo stato di Inizio Match serve a sincronizzare i due client (per l' esattezza serve a sincronizzare i giocatori – in quanto i tempi umani sono diversi dai tempi delle macchine e poi le macchine sono già state sincronizzate con il precedente stato di attesa – con un breve countdown prima di fare iniziare il match vero e proprio) e lo stato di Fine Match serve a gestire la vittoria o la sconfitta.

Inoltre un giocatore può decidere di abbandonare del tutto il gioco. Lo stato di abbandono serve a fare delle operazioni conclusive prima della terminazione del programma. E dal momento che può accadere che un giocatore abbandoni anche mentre sta interagendo con un altro giocatore (in fase di negoziazione o durante il match – per quanto sia scorretto) è bene includere uno stato di transizione con cui notificare al giocatore che invece è rimasto nel gioco che il suo avversario è fuggito e riportarlo in idle.

C' è da aggiungere un ulteriore aspetto, per quanto marginale, ma è bene discuterlo per completezza: dal momento che il client fa uso della rete per comunicare con il server (e viceversa) si dovrebbe prevedere uno stato per gestire eventuali cadute della rete successivamente alla fase di connesione. In realtà la questione è più teorica che pratica, dal momento che oggi giorno la presenza di una rete stabile è un dato di fatto ed è assai improbabile che possa verificarsi una caduta a meno

che non venga causata volontariamente (come scollegare di proposito i cavi o spegnere la wireless). Con la diffusione e la tecnologia di oggi, la probabilità che possa verificarsi una caduta della comunicazione è praticamente nulla. Tuttavia, per completezza, si introduce un ulteriore stato (comunicazione interrotta) tramite il quale garantire una terminazione soft del client, notificando all' utente l' impossibilità di continuare a comunicare. Come già detto, si tratta di un aspetto marginale, inoltre ha una ragion d' essere diversa da quella dello stato di "Gestione errore di connessione":

- "Gestione errore di connessione" serve a gestire problemi non dipendenti dall' applicazione all' atto della connessione, intesa come inizio della comunicazione tra client e server, problemi come aver sbagliato indirizzo del server o non essere fisicamente connesso alla rete. Per farla breve, questo è lo stato in cui il client si ritrova se, prima di iniziare a comunicare, si accorge che non può comunicare.

- "Comunicazione interrotta" serve a gestire la caduta della comunicazione in un qualunque momento successivo a quando è stata stabilita. Questo è lo stato in cui il client si ritrova se, in un momento successivo a quando ha iniziato a comunicare (con successo) con il server, si accorge che non può più farlo. Si tratta di uno stato più marginale e inutile rispetto al precedente perchè oggi giorno la rete è una certezza e le cadute sono molto improbabili, la rete, se c' è (e nello stato "Gestione errore di connessione" si controlla che ci sia) continua ad esserci e, a meno di non causare volontariamente un' interruzione fisica, continuerà ad esserci.

L' aspetto appena discusso è raffigurato nella figura 3.13b, in cui lo si integra con tutto il resto. Tuttavia, per semplicità, in seguito lo si ometterà volontariamente, facendo riferimento al grafico 3.13.

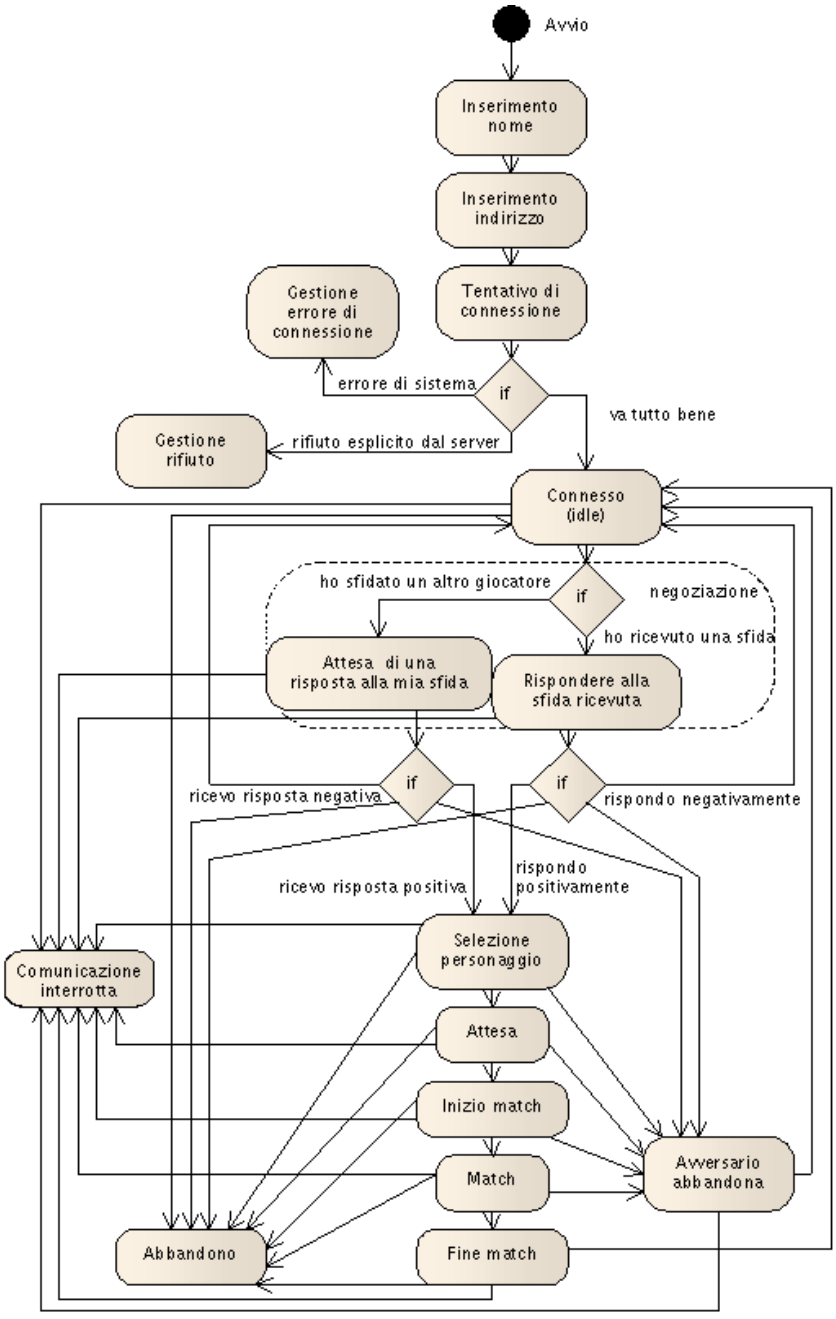

Figura 3.13b: Diagramma delle attività che illustra con maggiore dettaglio il comportamento del client

Per quanto riguarda l' aspetto dell' interazione è stato delineato già dai requisiti e ulteriormente chiarito in analisi quali sono gli scambi di informazione, quando avvengono e quali informazioni vengono scambiate ogni volta. Ma non è ancora del tutto sufficiente. Dunque si approfondisce maggiormente questo aspetto, con l' obiettivo soprattutto di dare una specifica dei messaggi maggiormente dettagliata. Si specificano i seguenti messaggi, qui distinti in due gruppi in base alla direzione:

- da client a server:
  - CONNESSIONE
  - SFIDA
  - RISPOSTA A SFIDA
  - PERSONAGGIO SCELTO
  - COMANDO
  - ABBANDONO
- da server a client:
  - ACCETTAZIONE CONNESSIONE
  - LISTA PARTECIPANTI
  - NOTIFICA SFIDA
  - NOTIFICA RISPOSTA
  - PRONTI VIA
  - STATO MATCH
  - FINE MATCH
  - AVVERSARIO ABBANDONA

Vediamo di seguito come questi vengono utilizzati durante una sessione di gioco. Si indicano con MC i messaggi inviati dal client e MS quelli dal server.

- (MC) All' atto di connettersi al server il client invia un messaggio di CONNESSIONE, che contiene il nickname del giocatore.

- (MS) Ricevuta la connessione del client il server risponde immediatamente con il messaggio ACCETTAZIONE CONNESSIONE per comunicare se può accettarla o no.

- (MS) Se il client viene accettato, il server gli invia subito la LISTA PARTECIPANTI, per comunicargli quanti altri giocatori sono connessi e come si chiamano.

- (MC) Quando un giocatore vuole sfidarne un altro, il proprio client invia al server un messaggio SFIDA contenente il nome del giocatore che desidera sfidare.

- (MS) A questo punto il server si fa carico di inoltrare la richiesta fino all' altro client, con un messaggio di NOTIFICA SFIDA

- (MC) Il giocatore che riceve la sfida può decidere di accettarla o rifiutarla. In entrambi i casi invierà al server la risposta per mezzo di un messaggio RISPOSTA A SFIDA

- (MS) E di nuovo il server inoltra la risposta all' altro client per mezzo di una NOTIFICA RISPOSTA. (Si è scelta quindi una soluzione in cui il server fa da intermediario)

- (MC) Ogni client invia al server il personaggio che ha selezionato con il messaggio PERSONAGGIO SCELTO.

- (MS) Quando entrambi i client hanno selezionato il proprio personaggio il server invia loro il PRONTI VIA per farli uscire dall' attesa e sincronizzarli.

- (MC) Ha così inizio il match tra i due giocatori che possono compiere azioni di due tipi: effettuare uno spostamento (avanti, indietro, destra e sinistra) o attaccare (con uno dei possibili colpi a scelta). Effettuando una di queste azioni il client invia al server un messaggio COMANDO con l' azione effettuata.

- (MS) Il server, alla ricezione, elabora tale messaggio, svolge i calcoli necessari e successivamente invia un aggiornamento dello stato del match ad entrambi con il messaggio STATO MATCH.

Questo messaggio riporta i valori (salute, forza, posizione, direzione, ecc) di entrambi i giocatori e va inviato ad entrambi perché tutti e due devono vedere la stessa cosa. Va anche inviato a chi ha effettuato l' azione: si supponga che il client A attacchi: il suo colpo potrebbe andare a segno, ma B potrebbe avere i riflessi pronti e pararlo, questo significa che non solo B deve sapere di essere stato colpito o no, ma anche A deve sapere se ha portato l' attacco a segno o meno.

- (MS) Il match finisce quando uno dei due sfidanti esaurisce tutta la propria salute. In tal caso viene inviato un messaggio FINE MATCH ad entrambi con cui comunicare vittoria o sconfitta.

- (MC) Il client di un giocatore connesso che abbandona il server gli invia un messaggio di ABBANDONO con cui comunica che se ne sta andando. Quando un giocatore se ne va non è più tra i partecipanti, pertanto è necessario inviare di nuovo la lista dei partecipanti a tutti i giocatori che sono rimasti per aggiornarli.

- (MS) Infine se un giocatore abbandona mentre sta interagendo con un altro giocatore (si stanno mettendo d' accordo se disputare un match o stanno già lottando) il server avvisa il giocatore rimasto con un messaggio AVVERSARIO ABBANDONA

La figura 3.14 riesce a sintetizzare e mostrare meglio l' ordine temporale delle interazioni, restando fedele a quanto precedentemente mostrato in figura 3.4 e al tempo stesso mostrando il maggior dettaglio dato dalla specifica dei messaggi appena riportata.

Manca solo da discutere la progettazione della parte server (figura 3.15): si era già detto che il server ha due aspetti da gestire insieme, il primo riguarda l' arrivo di nuovi giocatori (che si concretizza nella connessione di nuovi client) e la gestione dei giocatori al momento liberi, mentre il secondo riguarda la gestione dei client impegnati nei match. Si è deciso di fare in modo che la gestione dei match venga attivata dalla richiesta di una sfida.

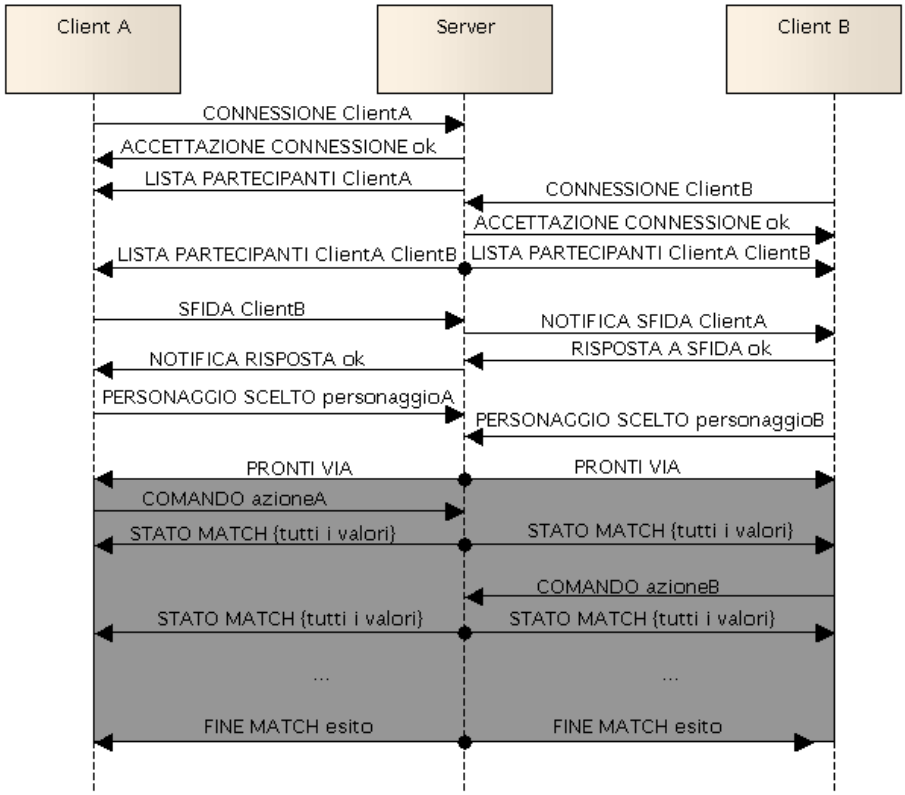

Figura 3.14: Diagramma delle interazioni maggiormente dettagliato

Si attiva un' attività parallela alla gestione di connessioni e client liberi. Questo significa che per ogni coppia di client impegnati è in esecuzione una specifica attività di gestione del match. Tale attività parte con la richiesta di una sfida e termina quando il match finisce (per ko o per abbandono da parte di uno dei due) o termina prematuramente se lo sfidato non accetta. In figura 3.15 si fa uso del costrutto fork per indicare l' avvio di una nuova attività. Il fork qui usato è da intendersi in un senso molto più vicino alla fork del C, in quanto si vuol indicare che, all' arrivo di una richiesta di sfida da parte di un client nasce una nuova attività di gestione dei client impegnati (ramo di destra), mentre la vecchia attività di gestione delle connessioni e dei client liberi continua normalmente (ramo di sinistra). Più che parallele in senso stretto, le due attività sono concorrenti.

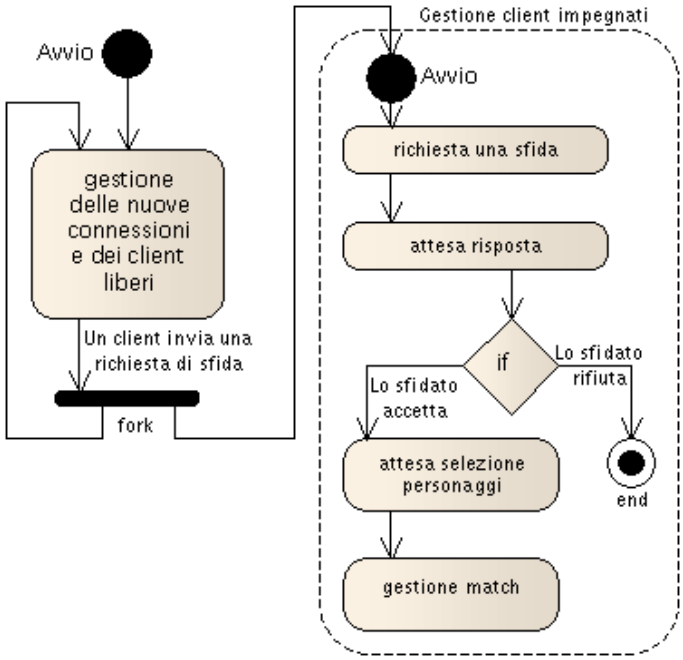

Figura 3.15: Diagramma delle attività che illustra con maggiore dettaglio il comportamento del server

A conclusione di questa sezione dedicata alla progettazione si riporta un diagramma (figura 3.16) che illustra come i comportamenti dei client e del server e le loro reciproche interazioni siano aspetti globalmente coerenti e collegati. Come nel caso del diagramma 3.7, si tratta di una rappresentazione che non è formalmente definita, ma è stata costruita appositamente unendo i vari aspetti, per gli stessi scopi precedentemente indicati. Sono riportate in orizzontale e colorate di rosso le interazioni (messaggi), in verticale la parte comportamentale di client e server e in blu l' evoluzione del comportamento.

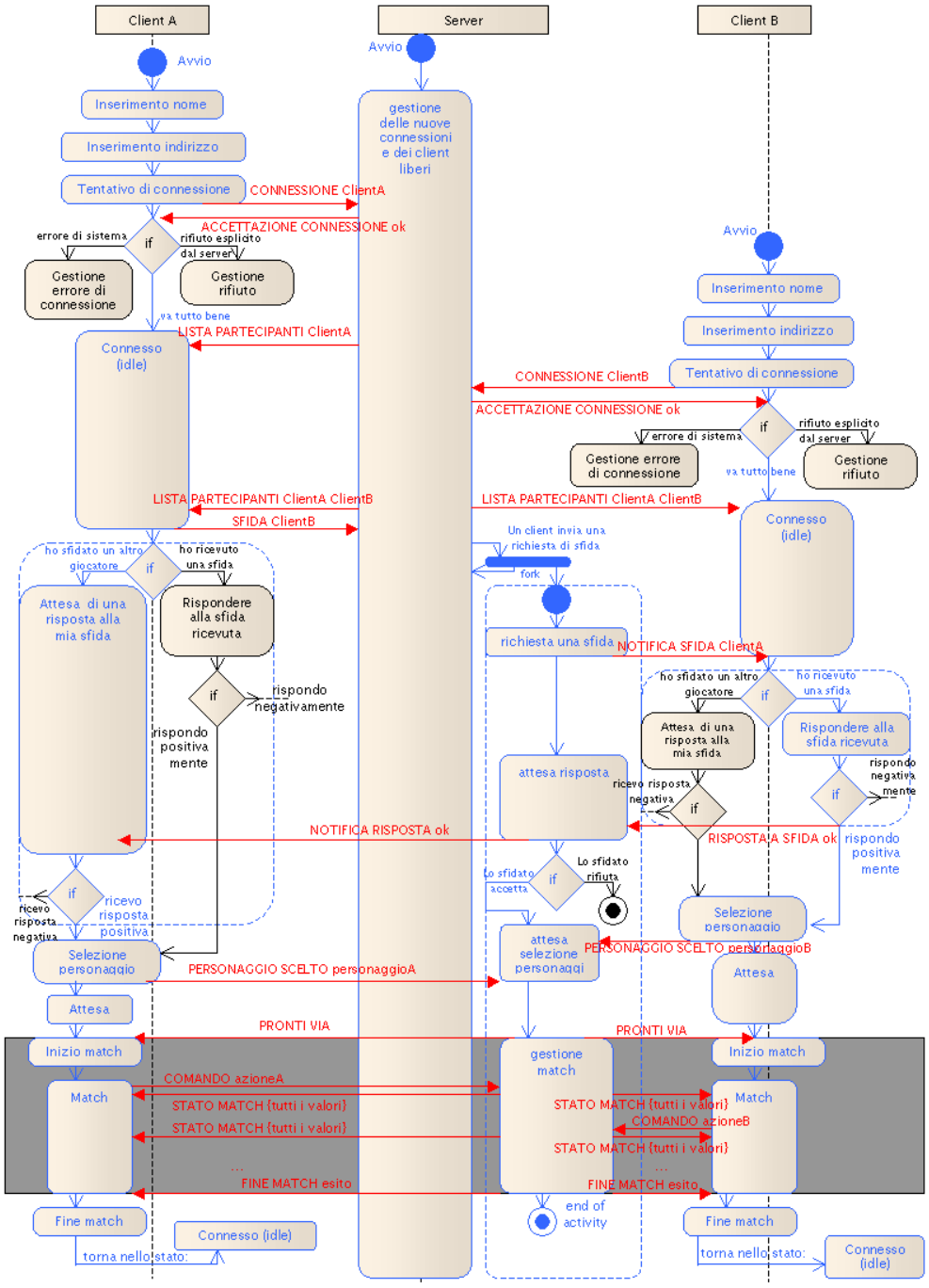

Figura 3.16: Visione complessiva e dettagliata dei comportamenti interni e delle interazioni tra due client e un server

Per completezza saranno velocemente illustrate le dinamiche per le due situazioni alternative alla riuscita della connessione. Nel primo caso (figura 3.17) si supponga che il server abbia la capacità di riuscire a gestire fino a N client contemporaneamente e si supponga che siano effettivamente connessi N client. Si ipotizzi che un client (N+1)esimo tenti di connettersi. La sua connessione sarà esplicitamente rifiutata dal server. Nella figura si riportano solo le parti significative.

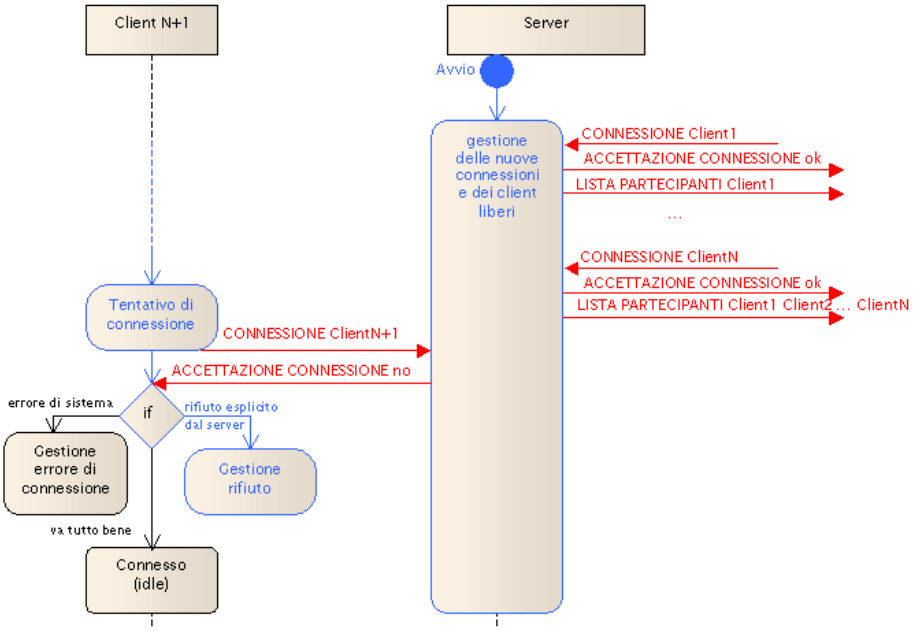

Figura 3.17: Connessione rifiutata dal server per raggiunto massimo numero di client

Il secondo caso è quello in cui un client tenta di connettersi al server ma qualcosa ad un livello inferiore non va come dovrebbe (ad esempio la rete non funziona) e il client evolve in una situazione di errore di sistema. In figura 3.18 si riportano solamente le parti significative.

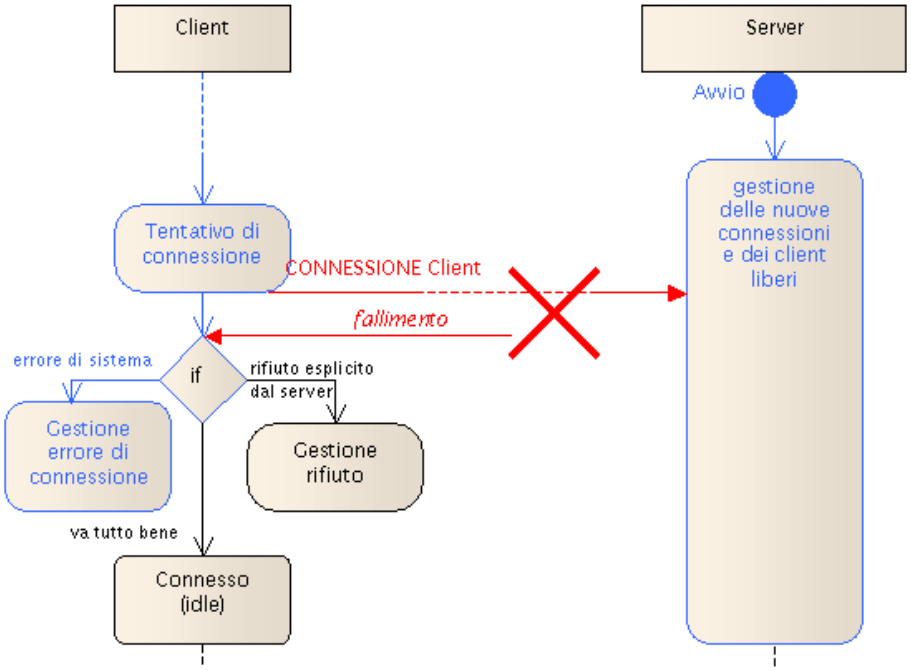

Figura 3.18: Connessione fallita per errori di sistema

3.4 Velocizzare il processo produttivo

Si propone di seguito una riflessione sul processo produttivo seguito, con l' intenzione di cercare una via per velocizzarlo. Questa non è tanto la necessità di uno sviluppatore occasionale, ma quella di una software house specializzata nel fare videogiochi.

La parte più delicata è quella relativa al client perché, dovendo andare su piattaforme diverse, è bene che sia altamente modulare così da poter essere riadattata velocemente per la specifica piattaforma. Problematiche platform-specific sono le operazioni di lettura da file (modelli 3d e texture devono essere caricati), la gestione degli input (mouse e tastiera, ma se si passa a consolle di gioco è diverso, così come è diverso se si passa a smartphone, nel quale si ha a che fare con il touchscreen) e tutto ciò che concerne l' aspetto grafico. Ci sono poi degli aspetti ancor più specifici che si sceglie di suddividere

ulteriormente per ragioni di semplicità e per poter poi permettere uno sviluppo incrementale del software finale. Tali aspetti sono la gestione degli input, per la parte grafica si suddivide ulteriormente il disegno degli elementi in scena e le scritte a schermo e inoltre si suddividono ulteriormente il loop di ricezione dei messaggi e l' idle loop dell' applicazione. Ma in che modo? Aver definito il comportamento del client con il grafo degli stati mostrato in figura 3.13 equivale a dire che il client ha 16 stati, intesi come modalità di funzionamento o anche fasi logiche del funzionamento, e sono:

- avvio
- inserimento nome
- inserimento indirizzo
- tentativo di connessione
- errore di connessione
- connessione rifiutata
- connesso (idle)
- rispondere a sfida
- attesa di una risposta
- selezione del personaggio
- attesa
- inizio match
- match
- fine match
- abbandono dell' avversario
- abbandono
- comunicazione interrotta

In questo caso aver suddiviso ulteriormente equivale a dire che, ad esempio, non si fa un' unica gestione degli input, ma questa è al suo interno divisa in più gestioni degli input, ciascuna relativa ad uno stato. Quando l' applicazione passa da uno stato ad un altro usa la gestione degli input relativa a quello stato se in tale stato è previsto di dover gestire degli input (come durante il match) o non fa niente se non è previsto (come nell' attesa). Potenzialmente possono esserci fino a 16 gestioni degli input, anche se in realtà ne servono di meno. E in maniera simile vale la stessa cosa per gli altri aspetti citati.

Il terzo aspetto è relativo alla comunicazione: sono stati definiti dei messaggi che vanno da client a server e viceversa. In questo caso la suddivisione è orientata non allo stato ma al messaggio: se si devono gestire dei messaggi in ingresso, piuttosto che gestire messaggi di qualunque tipo si è preferito avere un handler per ogni messaggio e in maniera simile una operazione (funzione o metodo) per inviare un certo specifico messaggio. A prima vista può sembrare uno spreco di tempo: suddividere così maniacalmente da progettare delle parti che poi ci si renderà conto che non sono necessarie dal punto di vista applicativo e dovranno essere eliminate. Eppure, riflettendoci bene si capisce che la struttura architetturale proposta non vale solo per il gioco "PunchingBalls", è più generale: in ogni gioco c' è una parte di gestione degli input, una per la grafica e se è pure distribuito, come in questo caso, c' è pure la parte per gestire la comunicazione e oltre a tutto questo "equipaggiamento" del client c' è anche il server, che non sarà troppo dissimile da quanto precedentemente illustrato. Quello che cambia davvero da un gioco all' altro è il comportamento interno di client e server e le loro reciproche interazioni e queste hanno influenza non tanto sulla struttura architetturale proposta, quanto nello specifico all' interno di alcune sue parti. Una soluzione elegante e pratica consisterebbe nel partire dalla specifica dei messaggi e degli stati per produrre in maniera automatica tutta l' infrastruttura e poi su questa aggiungere la parte più specifica inerente al gioco nella sua fattispecie: si avrebbe così un grande risparmio di tempo e di lavoro. Un framework del genere consentirebbe di avere la parte di analisi e di progettazione già svolte, in quanto le fissa una volta per tutte. Consentirebbe anche di disporre da subito e a costo zero di buona parte del codice del prodotto finale.

Per usare dei termini specifici dell' ingegneria del software si può dire che quello costruito e descritto in fase di progettazione non è solo il modello di PunchingBalls, ma discende da un meta-modello del quale il modello di PunchingBalls ne è un' istanza. Allo stesso modo il modello di un altro gioco diverso sarebbe anche questo un' altra istanza dello stesso meta-modello. E il framework ipotizzato poche righe sopra avrebbe la conoscenza di questo meta-modello.

3.5 Generazione automatica del codice

Si è optato di costruire un piccolo tool che potesse svolgere la generazione di codice a partire dalle specifiche descritte nell' ultima parte della precedente sezione. Tra le specifiche che prende in ingresso, vi è quella dei messaggi: da questa genera in automatico tutta la parte Network per Client e Server. Queste due parti sono concettualmente duali, nel senso che nella parte Network del Client si trova tutta la struttura necessaria per inviare i messaggi che il Client invierà poi al Server e per gestire i messaggi che il Server invia al Client, mentre nella corrispondente parte lato Server si trova tutta la struttura necessaria per inviare i messaggi che il Server invia al Client e per gestire i messaggi che il Client invia al Server. Invio e gestione della ricezione sono tra loro concettualmente diversi: la parte che invia messaggi può essere generata automaticamente tutta quanta, dal momento che si è deciso di concretizzarle in un insieme di procedure che hanno come argomenti esattamente le informazioni da inviare. Sarà poi il codice cosiddetto di "specific-logic" che dovrà solo effettuare una chiamata a tale procedura passando gli argomenti opportuni. Della parte di ricezione può essere prodotto in automatico la ricezione del messaggio dalla rete e l' estrazione da questo messaggio delle informazioni (quindi dei valori) che sono stati inviati.

Si consideri come esempio il codice generato lato Client per l' invio del messaggio di SELEZIONE avente il seguente prototipo:

```
void writeSELEZIONE(char* nome_giocatore, int char_id);
```

e la seguente implementazione, prodotta in automatico:

```
void writeSELEZIONE(char* nome_giocatore, int char_id){
      char SELEZIONE_string[255];
      bzero(SELEZIONE_string , 255);
      strcat(SELEZIONE_string, "SELEZIONE");
      strcat(SELEZIONE_string, " ");
      strcat(SELEZIONE_string, nome_giocatore);
      #define char_id_string_length 10
      char char_id_string[char_id_string_length];
      bzero(char_id_string, char_id_string_length);
      sprintf (char_id_string, " %d", char_id);
      strcat(SELEZIONE_string, char_id_string);
      writeShortLine(fd, SELEZIONE_string);
}
```

(fd è il file descriptor associato al socket connesso al server)

Lato Server viene prodotta in automatico un procedura che riceve il messaggio dalla rete e ne estrapola le informazioni significative. Solamente le ultima due righe (in blu) sono state aggiunte successivamente a mano in quanto si tratta di due righe di codice specifico per la gestione di queste informazioni:

```
void handleSELEZIONE(int index) {
      int tokenization_index = 10;
      int token_len = 0;
      // Recupero l' argomento nome_giocatore...
      char token_nome_giocatore[max_token_length];
      bzero(token_nome_giocatore, max_token_length);
      token_len = tokenizeMessage(msg_buff, tokenization_index,
      token_nome_giocatore);
      tokenization_index = tokenization_index + token_len + 1;
      /*EVENTUALI ALTRE ELABORAZIONI DEL TOKEN ESTRATTO...*/
      // Recupero l' argomento char_id...
      char token_char_id[max_token_length];
      bzero(token_char_id, max_token_length);
      token_len = tokenizeMessage(msg_buff, tokenization_index,
      token_char_id);
      tokenization_index = tokenization_index + token_len + 1;
      /*EVENTUALI ALTRE ELABORAZIONI DEL TOKEN ESTRATTO...*/
      /*CODIFICARE QUI DI SEGUITO IL COMPORTAMENTO DA ATTUARE...*/
      strcpy(ospiti[index] , token_nome_giocatore);
      characters_ospiti[index] = atoi(token_char_id);
}
```

Si nota subito una cosa: nella parte Network generata automaticamente – e solo in quella parte – si entra nei dettagli del protocollo di comunicazione di basso livello usato (in questo caso TCP). Al di fuori, invece, si usano le procedure qui generate. Questo significa che un altro vantaggio è che è possibile dimenticarsi completamente del protocollo di basso livello usato, perché la gestione della comunicazione a

questo livello è stata già scritta in automatico una volta per tutte. Gli aspetti di basso livello vengono mascherati. Tutto ciò che si deve fare per avere l' infrastruttura di comunicazione pronta per essere usata è solamente dare in ingresso al generatore una specifica dei messaggi di questo tipo:

```
CONNESSIONE                 C-->S char* nome_giocatore
SFIDA                       C-->S int id_avversario
RISPOSTA_A_SFIDA            C-->S int risposta
PERSONAGGIO_SCELTO          C-->S int char_id
COMANDO                     C-->S int cmd_id
ABBANDONO                   C-->S
ACCETTAZIONE_CONNESSIONE    S-->C int esito
LISTA_PARTECIPANTI          S-->C longlong id_update int numero_partecipanti int
      tuo_numero char* lista
NOTIFICA_SFIDA              S-->C int id_avversario char* nome_avversario
NOTIFICA_RISPOSTA           S-->C int risposta
INIZIO_MATCH                S-->C char* inizio_match_string
STATO_MATCH                 S-->C double pos_x double pos_y int aim int salute int
      forza int colpo int angolo_dx int angolo_sx longlong last_hit_time longlong
      last_rinco_time double avv_pos_x double avv_pos_y int avv_aim int avv_salute int
      avv_forza int avv_colpo int avv_angolo_dx int avv_angolo_sx longlong
      avv_last_hit_time longlong avv_last_rinco_time
FINE_MATCH                  S-->C int win_or_loose
NOTIFICA_ABBANDONO          S-->C
```

Ogni riga riporta la specifica di un messaggio, dichiarato con la seguente sintassi: (nome messaggio) (direzione, che può essere da Client a Server o quella contraria) lista di argomenti nel formato (tipo nome) Si è optato poi per ri-strutturare Network a livello architetturale: oltre a Client e Server è stato aggiunto un ulteriore blocco Common che al suo interno ha un LowLevelNet che sono due procedure di basso livello una per inviare e l' altra per ricevere dei byte. Dentro Client e Server la parte Network è stata ri-battezzata HighLevelNetwork in quanto si appoggia sul codice di basso livello. Il contenuto della HighLevelNetwork è esattamente quello descritto sopra. La figura 3.19 riporta tale riorganizzazione.

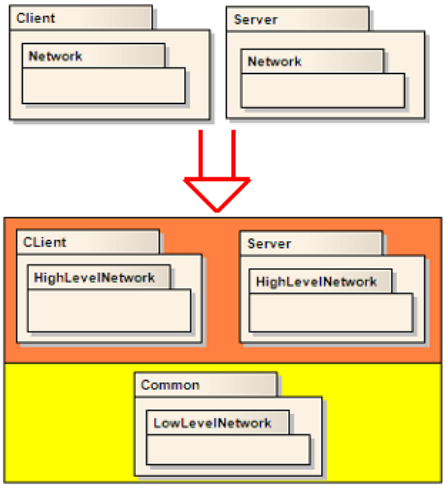

Figura 3.19: Ristrutturazione della parte Network in un layer di alto livello e uno di basso livello

Inoltre è possibile generare in automatico, lato server lo stato passivo della connessione per accettare i client, lato client lo stato attivo della connessione verso il server.

Specificando tutti gli stati di funzionamento – questa parte è stata supportata solo per il Client! – è possibile produrre automaticamente lo scheletro della gestione di input, disegno, idle loop e loop di ricezione. Purtroppo, essendo questa parte fortemente specific, di meglio non si può fare. Spetterà poi al programmatore riempire queste parti.

Un aspetto che è invece possibile "riciclare" è il motore grafico: basta solo portare fuori dal motore le parti più specifiche di disegno degli oggetti in scena (drawScene) e di stampa dei testi (printInterface) e chiamarle da dentro il motore per avere anche questa parte completamente riusabile.

```
void display(){
      glEnable(GL_DEPTH_TEST); // depth buffering abilitato...
      glShadeModel(GL_SMOOTH); // shading inizializzato...
      // Abilitazione texture e setting dei parametri...
      glEnable(GL_TEXTURE_2D);
      glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
      glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST);
      glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST);
      glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
      // Pulizia dei buffer...
      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
      glMatrixMode (GL_MODELVIEW);
      glLoadIdentity ();
      // Posizionamento della camera...
      setCamera();
      gluLookAt(eye[0], eye[1], eye[2], aim[0], aim[1], aim[2], up[0], up[1], up[2]);
      // Inizializzazione e accensione delle luci...
      glEnable(GL_LIGHTING);
      setLights();

      // Disegno degli oggetti presenti in scena
      drawScene();

      glDisable(GL_LIGHTING);
      glDisable (GL_TEXTURE_2D);
      glDisable (GL_DEPTH_TEST);
      glShadeModel (GL_FLAT);
      // Definizione dello spazio di visualizzazione
      glMatrixMode (GL_PROJECTION);
      glLoadIdentity ();
      GLdouble lef = -0.5;
      GLdouble rig = window_x-0.5;
      GLdouble bot = -0.5;
      GLdouble top = window_y-0.5;
      GLdouble nea = -1.0;
      GLdouble far = 1.0;
      glOrtho (lef, rig, bot, top, nea, far);
      glMatrixMode (GL_MODELVIEW);
      glLoadIdentity ();

      // Stampa delle eventuali scritte a schermo per creare l' interfaccia per l'
      utente...
      printInterface();

      reshape(window_x, window_y);

      glFlush();
      glutSwapBuffers();
}
```

Un altro aspetto ancora generabile in maniera automatica è legato alla gestione dei comandi (non degli input, ma di come gli input sono associati alle varie azioni) che è possibile produrre a partire da una piccola specifica del mapping dei comandi con i tasti. A partire dalla specifica di mapping tra tasti e azioni:

```
Mov_Avanti 101
Mov_Indietro 103
Mov_Destra 102
Mov_Sinistra 100
Att_Destro 0
Att_Sinistro 2
Att_Incrociato 127
Att_Acciaio 105
Att_Stordente 107
Att_Parata 1
```

viene generata la procedura loadKeyConfiguration per il caricamento dei comandi:

```
void loadKeyConfiguration(){
      // Leggo i comandi dal file...
      FILE * file;
      file =
      fopen("Client/UserInterfacing/ControlFunctions/configuration/keys.txt",
      "r");
      if (file == NULL){
            // Stampa comandi di default...
            printf("Impossibile accedere al file di impostazione dei
            comandi.\n");
            printf("Verranno usate le impostazioni di default:\n\n");
            printf(" Mov_Avanti:    101    (FrecciaSu)\n");
            printf(" Mov_Indietro:    103    (FrecciaGiù)\n");
            printf(" Mov_Destra:    102    (FrecciaDestra)\n");
            printf(" Mov_Sinistra:    100    (FrecciaSinistra)\n");
            printf(" Att_Destro:    0    (PulsMouseSinistro)\n");
            printf(" Att_Sinistro:    2    (PulsMouseDestro)\n");
            printf(" Att_Incrociato:    127    (Canc)\n");
            printf(" Att_Acciaio:    105    (Fine)\n");
            printf(" Att_Stordente:    107    (PagGiù)\n");
            printf(" Att_Parata:    1    (PulsMouseCentrale)\n");
            // Setta le impostazioni di default...
            key_Mov_Avanti = 101;
            key_Mov_Indietro = 103;
            key_Mov_Destra = 102;
            key_Mov_Sinistra = 100;
            key_Att_Destro = 0;
            key_Att_Sinistro = 2;
            key_Att_Incrociato = 127;
            key_Att_Acciaio = 105;
            key_Att_Stordente = 107;
            key_Att_Parata = 1;
            // Scrivi il file con le impostazioni di default...
            file =
            fopen("Client/UserInterfacing/ControlFunctions/configuration/keys.tx
            t", "w");
            fprintf(file, "Mov_Avanti 101\n");
            fprintf(file, "Mov_Indietro 103\n");
            fprintf(file, "Mov_Destra 102\n");
            fprintf(file, "Mov_Sinistra 100\n");
            fprintf(file, "Att_Destro 0\n");
            fprintf(file, "Att_Sinistro 2\n");
            fprintf(file, "Att_Incrociato 127\n");
```

```
            fprintf(file, "Att_Acciaio 105\n");
            fprintf(file, "Att_Stordente 107\n");
            fprintf(file, "Att_Parata 1\n");
            fclose(file);
      } else {
            // Carica i comandi...
            fscanf(file, "Mov_Avanti %d\n",     &key_Mov_Avanti);
            fscanf(file, "Mov_Indietro %d\n",   &key_Mov_Indietro);
            fscanf(file, "Mov_Destra %d\n",     &key_Mov_Destra);
            fscanf(file, "Mov_Sinistra %d\n",   &key_Mov_Sinistra);
            fscanf(file, "Att_Destro %d\n",     &key_Att_Destro);
            fscanf(file, "Att_Sinistro %d\n",   &key_Att_Sinistro);
            fscanf(file, "Att_Incrociato %d\n",
      &key_Att_Incrociato);
            fscanf(file, "Att_Acciaio %d\n",    &key_Att_Acciaio);
            fscanf(file, "Att_Stordente %d\n",  &key_Att_Stordente);
            fscanf(file, "Att_Parata %d\n",     &key_Att_Parata);
            fclose(file);
      }
      cmd_array[0] = key_Mov_Avanti;
      cmd_array[1] = key_Mov_Indietro;
      cmd_array[2] = key_Mov_Destra;
      cmd_array[3] = key_Mov_Sinistra;
      cmd_array[4] = key_Att_Destro;
      cmd_array[5] = key_Att_Sinistro;
      cmd_array[6] = key_Att_Incrociato;
      cmd_array[7] = key_Att_Acciaio;
      cmd_array[8] = key_Att_Stordente;
      cmd_array[9] = key_Att_Parata;
}
```

L' utilità di questa cosa è di dare la possibilità (anche se in questo caso non sfruttata) di inserire nel gioco una schermata di personalizzazione dei comandi in cui il mapping dei comandi viene salvato su file e ricaricato ai successivi avvii.

Tornando infine alla parte di grafica, si è scelto di usare dei formati per i modelli 3d e per le texture, questi formati sono lo Stanford (.ply) per i modelli 3d e le immagini .raw per le texture. Se si decide di usare tali formati anche la parte di caricamento e gestione di modelli 3d e texture può essere riusata – in quanto compresa nel codice generato in automatico.

3.6 Generazione di PunchingBalls Multiplayer

È ora di mettere in pratica quanto descritto. Nel file SpecificaKeys.txt copiamo la precedente specifica dei comandi: (un comando per ogni riga)

```
Mov_Avanti 101
Mov_Indietro 103
Mov_Destra 102
Mov_Sinistra 100
Att_Destro 0
Att_Sinistro 2
Att_Incrociato 127
Att_Acciaio 105
Att_Stordente 107
Att_Parata 1
```

Nel file SpecificaStatiClient.txt copiamo tutti gli stati del Client, già discussi precedentemente: (uno stato per ogni riga)

```
ST01_AVVIO
ST02_INSERIMENTO_NOME
ST03_INSERIMENTO_INDIRIZZO
ST04_TENTATIVO_DI_CONNESSIONE
ST05_ERRORE_DI_CONNESSIONE
ST06_CONNESSIONE_RIFIUTATA
ST07_CONNESSO_IDLE
ST08_NEGOZIAZIONE_SFIDATO_RISPONDERE
ST09_NEGOZIAZIONE_SFIDANTE_ATTESA
ST10_SELEZIONE_PERSONAGGIO
ST11_ATTESA
ST12_INIZIO_MATCH
ST13_MATCH
ST14_FINE_MATCH
ST15_AVVERSARIO_ABBANDONA
ST16_ABBANDONO
ST17_COMUNICAZIONE_INTERROTTA
```

Nel file SpecificaMessaggi.txt copiamo la specifica dei messaggi, già precedentemente mostrata:

```
CONNESSIONE                 C-->S char* nome_giocatore
SFIDA                       C-->S int id_avversario
RISPOSTA_A_SFIDA            C-->S int risposta
PERSONAGGIO_SCELTO          C-->S int char_id
COMANDO                     C-->S int cmd_id
ABBANDONO                   C-->S
ACCETTAZIONE_CONNESSIONE    S-->C int esito
LISTA_PARTECIPANTI          S-->C longlong id_update int numero_partecipanti int
      tuo_numero char* lista
NOTIFICA_SFIDA              S-->C int id_avversario char* nome_avversario
NOTIFICA_RISPOSTA           S-->C int risposta
INIZIO_MATCH                S-->C char* inizio_match_string
STATO_MATCH                 S-->C double pos_x double pos_y int aim int salute int
      forza int colpo int angolo_dx int angolo_sx longlong last_hit_time longlong
      last_rinco_time double avv_pos_x double avv_pos_y int avv_aim int avv_salute int
      avv_forza int avv_colpo int avv_angolo_dx int avv_angolo_sx longlong
      avv_last_hit_time longlong avv_last_rinco_time
FINE_MATCH                  S-->C int win_or_loose
NOTIFICA_ABBANDONO          S-->C
```

È necessario qualche ulteriore dettaglio. Nel file specs.txt copiamo "la specifica delle specifiche" ovvero in quali file si trovano le varie specifiche. Sarà infatti questo ultimo file che andrà in ingresso al generatore di codice (e funzionerà un po' come un puntatore)

```
(Nome_della_directory-progetto_da_crare.............)
      PunchingBallsMultiplayer
(Nome_del_file_di_specifica_degli_stati_del_client..)
      PBMspec/SpecificaStaticlient.txt
(Nome_del_file_di_specifica_dei_comandi_(keys)_.....)
      PBMspec/SpecificaKeys.txt
(Nome_del_file_di_specifica_dei_messaggi............)
      PBMspec/SpecificaMessaggi.txt
(Specificare_la_porta_del_server....................)
      31213
```

I dettagli aggiunti riguardano il nome della cartella destinazione e la porta del Server, che deve essere fissata e decisa una volta per tutte (well known port).

E adesso tutto ciò che si deve fare è lanciare il generatore di codice passandogli come ingresso il percorso in cui si trova il file di "specifica delle specifiche".

A questo punto è possibile iniziare a lavorare aggiungendo al codice già prodotto e organizzato nella cartella "PunchingBallsMultiplayer" tutta la parte specifica di PunchingBalls Multiplayer, quegli aspetti di game-play che il generatore non può produrre in quanto è ciò che l' applicazione ha di unico e particolare.

Capitolo 4

Realizzazione del gioco su multipiattaforma

Le fasi più teoriche dello sviluppo (dalla definizione dei requisiti, poi l' analisi, fino alla progettazione) dell' applicazione proposta sono state illustrate nel capitolo precedente. Ma, dal momento che le fasi finali sono quelle più pratiche, (si scrive del codice in un linguaggio che verrà eseguito all' interno di un certo ambiente, in un sistema operativo caratterizzato da concetti e astrazioni proprie e legate alla macchina sottostante) in questo capitolo si approfondiscono delle questioni tecniche legate alla sfera del "come si fa", avendo cura di descrivere come costruire, nei vari ambienti, l' applicazione presentata.

4.1 Come fare un' applicazione OpenGL in C su pc

Per ragioni di completezza questa prima parte è dedicata a come si scrive un' applicazione OpenGL su pc. Tuttavia, essendo il computer un mondo molto familiare, la trattazione sarà molto breve. L' ambiente di riferimento è il sistema operativo Linux, scelto per ragioni di "apertura": si tratta di un mondo completamente open-source in cui tutto ciò che è disponibile, lo è a costo zero. Il linguaggio scelto è il C, per ragioni di semplicità, soprattutto se paragonato con l' altro possibile linguaggio, il C++. Si assume che si sappia come costruire

un' applicazione C. Si illustra invece come costruire un' applicazione OpenGL in C. Molto brevemente, per produrre un' animazione grafica che sia anche reattiva agli input da mouse e tastiera, si sfrutta OpenGL e la libreria GLUT. Questo perchè OpenGL è di solo output, nel senso che serve per fare resa grafica, ma non gestisce gli input dalle periferiche: gli ideatori volevano rendere OpenGL indipendente dal tipo di sistema, pertanto è stata pensata solo per la resa, mentre la gestione degli input, essendo specifica del sistema, dovrà essere demandata a librerie di più alto livello. Nel nostro caso questa libreria è la GLUT (GL Utility Tooltik), costruita sopra GLU (GL Utility), che a sua volta è costruita sopra OpenGL. GLUT mette a disposizione di OpenGL il meccanismo delle callback. Per ogni aspetto che si desidera prendere in considerazione esiste una callback. Se si vuole rispondere agli input da tastiera sarà sufficiente implementare una propria callback. Perché la callback venga effettivamente eseguita è necessario prima "registrarla". Questo significa che il main dell' applicazione è diviso in una prima parte di inizializzazione in cui avviene anche la registrazione delle callback e una seconda parte, data dal loop grafico vero e proprio, che corrisponde al funzionamento.

```
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
glutInitWindowSize(window_x, window_y);
glutCreateWindow("PunchingBalls Multiplayer");
// Registro le callback...
glutReshapeFunc(reshape);     // gestisce ridimensionamento finestra
glutDisplayFunc(display);     // disegna un frame
glutMouseFunc(mouse);         // gestisce i tasti del mouse
glutKeyboardFunc(keys);       // gestisce una parte della tastiera
glutSpecialFunc(specialKeys); // gestisce i tasti speciali della tastiera
glutIdleFunc(idle);           // si usa per animare

glutMainLoop();
```

Figura 4.1: Il main di un' applicazione OpenGL in C

La figura 4.1 mostra un possibile main in cui si ha prima la registrazione delle varie callback e poi l' avvio del loop grafico. Da notare che anche la funzione di disegno (display) è trattata come una qualunque callback. La figura 4.2 mostra poi una possibile implementazione di callback.

```
void reshape(int w, int h){
      window_x = w;
      window_y = h;
      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
      // Piani di taglio...
      GLdouble left = -1.5;
      GLdouble right = 1.5;
      GLdouble bottom = -1.5;
      GLdouble top = 1.5;
      GLdouble near = 2.0;
      GLdouble far = 20.0;
      // Definizione esplicita del frustum...
      if( w<=h ) {
            GLdouble r = ((GLdouble)(h)) / ((GLdouble)(w));
            glFrustum(left, right, bottom*r, top*r, near, far);
      } else {
            GLdouble r = ((GLdouble)(w)) / ((GLdouble)(h));
            glFrustum(left*r, right*r, bottom, top, near, far);
      }
      glMatrixMode(GL_MODELVIEW);
      glLoadIdentity ();
      glViewport(0, 0, (GLint)w, (GLint)h);
}
```

Figura 4.2: L' implementazione di una callback. In particolare una callback per gestire il ridimensionamento della finestra.

4.2 "Mondo" Android

Android e iOS (iphone) sono i due ambienti leader nel settore degli smartphone. Non sono le uniche, ad esempio Windows Mobile ha una sua fetta di mercato, così come ancora Symbian non è del tutto scomparso. Tuttavia il grosso del mercato è concentrato su Android e iOS. La differenza nella distribuzione sta nel fatto che iOS è legato ai vari iphone e ipad, quindi un insieme piuttosto ristretto di dispositivi, i quali hanno ciascuno grandi fette di vendita, e hardware e sistema operativo sono della stessa casa produttrice. Invece Android è il sistema operativo di Google e inizialmente pensato solo per lo smartphone della stessa Google, ma successivamente montato sui dispositivi di vari produttori, tra i quali LG, Sony e soprattutto Samsung. In questo caso siamo di fronte ad una gamma di dispositivi più vasta, ciascuno con le proprie fette di vendita, ma aggregando le quali Android riesce a tener

testa ad iOS. Diversa è la politica attuata dai due giganti del settore: iOS è un mondo più chiuso e protetto rispetto ad Android, disponibile a costo zero per uno sviluppatore software. Gli sviluppatori di "app", oltre ai costi, incontrano nel mondo iOS anche maggiori limitazioni e barriere rispetto al mondo Android. Costi ridotti e accesso più facilitato sono i motivi per i quali è stato scelto Android come sistema operativo target per sviluppare l' applicazione su smartphone. Prima di addentrarci nella trattazione del "come si fa" è bene esaminare per qualche istante la struttura di Android. Fondamentalmente si tratta di un kernel Linux organizzato per gestire dispositivi di quel tipo, al di sopra del quale è installato un layer java. Java è il linguaggio di riferimento per la programmazione in ambiente Android. Data la natura del dispositivo fisico e le limitate risorse hardware, l' ambiente di esecuzione java è diverso da quello conosciuto per pc. Diversa è la macchina virtuale usata, non è la classica stack-based Java Virtual Machine, ma una più performante e ottimizzata register-based Dalvick Virtual Machine (Dalvick, dal nome del suo ideatore). In realtà la struttura di Android è un po' più complessa, ma per un neofita questo è più che sufficiente. Per ulteriori dettagli si rimanda alla figura 4.3. Come si evince dalla figura, Android attinge a piene mani dal mondo open source. Il cuore di ogni sistema Android, tanto per cominciare, è un kernel Linux, versione 2.6. Direttamente nel kernel sono inseriti i driver per il controllo dell' hardware del dispositivo: driver per la tastiera, lo schermo, il touch screen, il Wi-Fi, il Bluetooth, il controllo dell' audio e così via. Sopra il kernel poggiano le librerie fondamentali, anche queste tutte mutuate dal mondo Open Source. Da citare sono senz' altro OpenGL, per la grafica, SQLite, per la gestione dei dati, e WebKit, per la visualizzazione delle pagine Web. L' architettura prevede poi una macchina virtuale ed una libreria fondamentale che, insieme, costituiscono la piattaforma di sviluppo per le applicazioni Android. Questa macchina virtuale è la Dalvik, come già detto. Nel penultimo strato dell' architettura è possibile rintracciare i gestori e le applicazioni di base del sistema. Ci sono gestori per le risorse, per le applicazioni installate, per le telefonate, il file system ed altro ancora: tutti componenti di cui difficilmente si può fare a meno. Infine, sullo strato più alto dell' architettura, poggiano gli applicativi destinati all' utente finale. Molti, naturalmente, sono già inclusi con l'

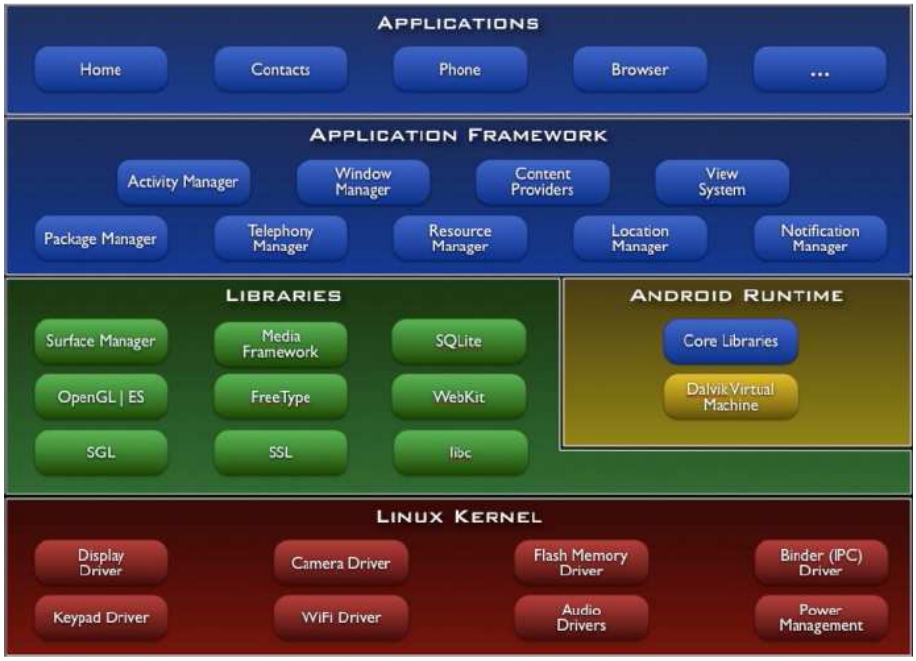

Figura 4.3: Architettura di Android.

installazione di base: il browser ed il player multimediale sono dei facili esempi. A questo livello si inseriranno anche le applicazioni, come quella sviluppata in questa tesi

Recentemente, oltre al classico SDK basato su java, Google ha distribuito un ambiente di programmazione alternativo, a più basso livello, noto come NDK, dove la N sta per Native e supporta i linguaggi nativi del kernel Linux, che sono C e C++. Tuttavia, stando alla documentazione ufficiale fornita da Google scrivere un' app con NDK piuttosto che con SDK permette di ottenere un guadagno di prestazioni molto basso, rendendo tuttavia più complicata l' applicazione. Inoltre, essendo NDK molto più recente rispetto a SDK, uno sviluppatore SDK in difficoltà potrebbe facilmente trovare la risposta ai suoi problemi in rete con pochi click, grazie alla presenza di una numerosa comunità, mentre la stessa cosa non è vera per NDK. A questo aggiungiamo il fatto che, trattandosi di un videogame, lo schermo non sarà riempito soltanto dal disegno di una scena, ma sarà

arricchito dalla presenza di scritte, pulsanti e altri elementi che vengono completamente forniti nelle apposite librerie java e potranno essere usati in una maniera molto simile a quella vista in swing - invece che usare classi di javax.swing si useranno classi di android.widget e android.view. Per finire, per la resa di scene tridimensionali, le librerie grafiche sfruttano l' hardware presente per effettuare le operazioni necessarie. Le librerie grafiche java non sono delle implementazioni a sé stanti, ma dei binding delle librerie native e questo significa che nel passare dal livello nativo del codice C al livello applicativo del codice java la perdita di prestazioni è molto contenuta. Dopo aver esaminato la lista dei pro e dei contro, la scelta è ricaduta sull' uso dell' ambiente di programmazione SDK (e quindi sulla programmazione in linguaggio java).

4.2.1 Come si fa un' applicazione Android

Il modo più semplice per iniziare è quello di avvalersi di un IDE (il più noto e usato è Eclipse con l' ADT plug-in): "Nuovo", "Progetto Android", si inserisce il titolo e un altro paio di nomi e si è pronti per iniziare. Chi è più esperto può avvalersi della CLI dell' SDK di Android (comando "android create project ... –activity PunchingBallsMultiplayer ..."). In entrambi i casi si popolerà automaticamente il progetto, inserendo le librerie di Android e la struttura di base dei progetti per questa piattaforma. Si avrà anche a disposizione la prima classe, già predisposta, in questo caso "PunchingBallsMultiplayerActivity", da cui partire per sviluppare l' applicazione. A cosa serve questa prima classe? E' presto detto. Dal momento che è stata precedentemente citata swing, realizzare un' applicazione java con swing implica estendere un JFrame. Se invece si sviluppano applicazioni java che girano nel browser, si deve cominciare da una Applet. Tutto questo per dire che ciascun ambiente, java e non, dispone dei suoi mattoni fondamentali, che lo sviluppatore può estendere ed implementare per trovare un punto di aggancio con la piattaforma. Android non sfugge alla regola, anzi la amplifica. A seconda di quel che si intende fare è disponibile un diverso modello. Android fornisce quattro mattoni di base:

- Attività: le attività sono quei blocchi di un' applicazione che interagiscono con l' utente utilizzando lo schermo ed i dispositivi di input messi a disposizione dallo smartphone. Comunemente fanno uso di componenti UI già pronti, come quelli presenti nel pacchetto android.widget, ma questa non è necessariamente la regola. Le attività sono probabilmente il modello più diffuso in Android, e si realizzano estendendo la classe base android.app.Activity.

- Servizio: un servizio gira in sottofondo e non interagisce direttamente con l' utente. Ad esempio può riprodurre un brano MP3, mentre l' utente utilizza delle attività per fare altro. Un servizio si realizza estendendo la classe android.app.Service.

- Broadcast Receiver: un Broadcast Receiver viene utilizzato quando si intende intercettare un particolare evento, attraverso tutto il sistema. Ad esempio lo si può utilizzare se si desidera compiere un' azione quando si scatta una foto o quando parte la segnalazione di batteria scarica. La classe da estendere è android.content.BroadcastReceiver.

- Content Provider: i Content Provider sono utilizzati per esporre dati ed informazioni. Costituiscono un canale di comunicazione tra le differenti applicazioni installate nel sistema. Si può creare un Content Provider estendendo la classe astratta android.content.ContentProvider.

Un' applicazione Android è costituita da uno o più di questi elementi. Molto frequentemente, contiene almeno un' attività, ma non è detto che debba sempre essere così. Nel nostro caso è costituita da un' Attività soltanto, la già citata PunchingBallsMultiplayerActivity. A conti fatti, le attività (Activity), sono il più fondamentale dei componenti di base delle applicazioni Android.

Stando alla documentazione ufficiale, un' attività è "una singola e precisa cosa che l' utente può fare". Proviamo ad indagare le implicazioni di questa affermazione. Partiamo dal fatto che l' utente, per fare qualcosa, deve interagire con il dispositivo. Domandiamoci come avvenga, nel caso di uno smartphone, l' interazione tra l' uomo e la

macchina. Un ruolo essenziale, naturalmente, è svolto dai meccanismi di input come la tastiera ed il touchscreen, che permettono all' utente di specificare il proprio volere. Le periferiche di input, tuttavia, da sole non bastano. Affinché l' utente sappia cosa può fare e come debba farlo, ma anche affinché il software possa mostrare all' utente il risultato elaborato, è necessario un canale aggiuntivo. Nella maggior parte dei casi questo canale è il display. Nella superficie dello schermo il software disegna tutti quegli oggetti con cui l' utente può interagire (bottoni, menù, campi di testo), e sempre sullo schermo viene presentato il risultato dell' elaborazione richiesta. Il ragionamento ci porta alla conclusione che, per fare qualcosa con il dispositivo, è necessario usare lo schermo. Esiste perciò un parallelo tra il concetto di attività, in Android, e quello di finestra, in un sistema desktop, benché non siano esattamente la stessa cosa. In generale, ad ogni modo, possiamo assumere con tranquillità che le attività sono quei componenti di un' applicazione Android che fanno uso del display e che interagiscono con l' utente. Dal punto di vista del programmatore, poi, possiamo spingerci oltre e semplificare ulteriormente. In maniera più pragmatica, un' attività è una classe che estende android.app.Activity. L' autore del codice, realizzando l' attività, si serve dei metodi ereditati da Activity per controllare cosa appare nel display, per assorbire gli input dell' utente, per intercettare i cambi di stato e per interagire con il sistema sottostante.

Si ritiene ora opportuno spendere qualche parola in merito al ciclo di vita di un' attività. In un sistema desktop il monitor è sufficientemente spazioso da poter mostrare più finestre simultaneamente. Perciò non è affatto raro lavorare con più programmi contemporaneamente attivi, le cui finestre vengono affiancate o sovrapposte. Gli smartphone, invece, funzionano diversamente. Prima di tutto il display è piccolo, e pertanto ha poco senso affiancare due o più finestre di applicazioni differenti. Poi non bisogna dimenticare che le risorse di calcolo sono modeste, e perciò non è buona cosa tenere simultaneamente in vita troppi programmi. Per questi motivi le attività di Android hanno carattere di esclusività. È possibile mandare in esecuzione più attività simultaneamente, ma soltanto un' attività alla volta può occupare il display. L' attività che occupa il display è in esecuzione ed interagisce direttamente con l' utente. Le altre, invece, sono ibernate e tenute

nascoste in sottofondo, in modo da ridurre al minimo il consumo delle risorse di calcolo. L' utente, naturalmente, può ripristinare un' attività ibernata e riprenderla da dove l' aveva interrotta, riportandola in primo piano. L' attività dalla quale si sta allontanando, invece, sarà ibernata e mandata in sottofondo al posto di quella ripristinata. Il cambio di attività può anche avvenire a causa di un evento esterno. Il caso più ricorrente è quello della telefonata in arrivo: se il telefono squilla mentre si sta usando la calcolatrice, quest' ultima sarà automaticamente ibernata e mandata in sottofondo. L' utente, conclusa la chiamata, potrà richiamare l' attività interrotta e riportarla in vita, riprendendo i calcoli esattamente da dove li aveva interrotti.

Siccome le attività ibernate, in termini di risorse di calcolo, non consumano nulla, in Android il concetto di chiusura delle attività è secondario e tenuto nascosto all' utente. Ciò, di solito, spiazza chi è al suo primo confronto con la programmazione dei dispositivi portatili. Le attività di Android non dispongono di un bottone "x", o di un tasto equivalente, con il quale è possibile terminarle. L' utente, di conseguenza, non può chiudere un' attività, ma può solo mandarla in sottofondo. Questo, comunque, non significa che le attività non muoiano mai, anzi! Per prima cosa le attività possono morire spontaneamente, perché hanno terminato i loro compiti. Insomma, anche se il sistema non ci fornisce automaticamente un bottone "chiudi", possiamo sempre includerlo noi nelle nostre applicazioni. In alternativa, la distruzione delle attività è completamente demandata al sistema. I casi in cui un' attività può terminare sono due: l' attività è ibernata ed il sistema, arbitrariamente, decide che non è più utile e perciò la distrugge, oppure il sistema è a corto di memoria, e per recuperare spazio inizia ad uccidere bruscamente le attività in sottofondo. Esistono poi dei task manager di terze parti che permettono di uccidere le attività in sottofondo, ma non sono previsti nel sistema di base. I differenti passaggi di stato di un' attività attraversano alcuni metodi della classe Activity che, come programmatori, possiamo ridefinire per intercettare gli eventi di nostro interesse.

La figura 4.4 illustra la sequenza di chiamate ai metodi di Activity eseguite durante i passaggi di stato dell' attività. Entriamo nel dettaglio:

- onCreate()

Richiamato non appena l' attività viene creata.

- onRestart()

  Richiamato per segnalare che l' attività sta venendo riavviata dopo essere stata precedentemente arrestata.

- onStart()

  Richiamato per segnalare che l' attività sta per diventare visibile sullo schermo.

- onResume()

  Richiamato per segnalare che l' attività sta per iniziare l' interazione con l' utente.

- onPause()

  Richiamato per segnalare che l' attività non sta più interagendo con l' utente.

- onStop()

  Richiamato per segnalare che l' attività non è più visibile sullo schermo.

- onDestroy()

  Richiamato per segnalare che l' applicazione sta per essere terminata.

La prassi richiede che, come prima riga di codice di ciascuno di questi metodi, si richiami l' implementazione di base (super) del metodo che si sta ridefinendo. È importante non dimenticare questa regola, altrimenti le attività sviluppate potrebbero non funzionare correttamente.

- onBackPressed()

Il metodo onBackPressed() della classe Activity può essere ridefinito dallo sviluppatore per intercettare la pressione del tasto "back" del dispositivo durante l' esercizio dell' attività. Per default il tasto "back" causa la chiusura dell' Activity corrente ed il ritorno a quella

precedente (o alle schermate di sistema). Tuttavia ogni applicazione ed ogni attività sono libere di cambiare questo comportamento e decidere quale funzionalità assegnare al tasto "back". Nel far ciò, è importante che il tasto risulti comunque coerente con la logica dell' applicazione ed intuitivo da utilizzare. Ad esempio nel browser di sistema il tasto "back" viene intercettato ed usato per fare "indietro" nella cronologia di navigazione delle pagine Web. Solo quando si arriva in fondo alla cronologia, e pertanto non ci sono più pagine su cui retrocedere, allora un' ulteriore pressione del tasto "back" causa la chiusura dell' attività. Si tratta di un modello coerente, funzionale ed usabile. Nel nostro caso il metodo onBackPressed() viene usato per sostituire la pressione del tasto Esc su pc.

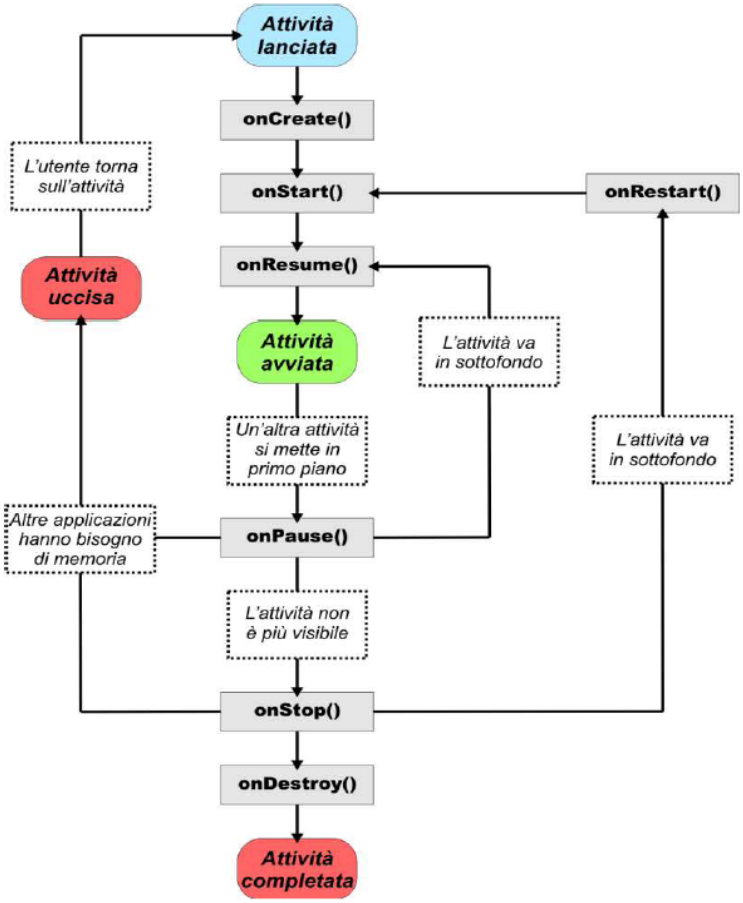

Figura 4.4: Ciclo di vita delle Attività Android.

4.2.2 Come si fa un' applicazione OpenGL ES in Android

OpenGL ES è uno standard industriale per la programmazione grafica 3D su dispositivi mobile. Khronos Group, un conglomerato che include marchi come ATI, NVIDIA ed Intel si preoccupa di definire ed estendere lo standard. Attualmente esistono 3 versioni di OpenGL ES: 1.0, 1.1 e 2.0: la versione scelta è stata la 1.0 poiché supportata dalla quasi totalità dei dispositivi Android e dall' emulatore rilasciato insieme all' ambiente di sviluppo. Per produrre della grafica tridimensionale su un dispositivo Android si fa riferimento in particolare a due classi fondamentali: una è GLSurfaceView, l' altra è GLSurfaceView.Renderer. Il rapporto tra le due è che GLSurfaceView è l' oggetto che va incluso nel layout della propria Activity ed è l' oggetto grafico sul quale il GLSurfaceView.Renderer andrà a disegnare.

Comune alle applicazioni Android è la disposizione di componenti (detti widget) nel layout visualizzato dalla propria Activity, componenti quali campi di testo, pulsanti, bottoni circolari, ecc. Nel caso della grafica tridimensionale il componente da includere è un GLSurfaceView o meglio ancora un proprio oggetto che estenda tale classe (android.opengl.GLSurfaceView). Il GLSurfaceView è poco più di un contenitore e di per sé è inutile se non gli viene associato un GLSurfaceView.Renderer (o, con maggior esattezza, un proprio oggetto che implementi l' interfaccia GLSurfaceView.Renderer) e che sarà il vero responsabile di ciò che verrà visualizzato sullo schermo del proprio smartphone. Programmare il proprio Renderer significa fondamentalmente implementare tre metodi:

- onSurfaceCreated()

  Chiamato una sola volta, al momento della costruzione del Renderer, da usare per caricare dati esterni (come i modelli poligonali e l texture) o per settare opzioni permanenti

- onSurfaceChanged()

  Chiamato tutte le volte che "la superficie cambia", e questo significa sia la prima volta al momento della costruzione del renderer, sia in caso di rotazione dello schermo. Lo si usa per ridefinire la matrice di wieport in una di queste circostanze.

- onDrawFrame()

  Questo è il metodo con il quale si disegna un singolo frame. Tutto ciò che riguarda il disegno degli elementi sulla scena va fatto qui. Questo significa che in questo metodo si maneggia la camera, si posizionano le sorgenti luminose e gli oggetti da disegnare. Non essendoci niente per la gestione esplicita delle animazioni, se si vuole produrre una scena dinamica lo si dovrà fare sempre in questo metodo. Questa mancanza contribuisce a "sporcare concettualmente" il codice, in quanto in questo stesso metodo si ha a che fare con l' aspetto statico di disegno del singolo frame e con l' aspetto dinamico dell' animazione, che copre più frame. Nella versione di OpenGL esaminata all' inizio di questo capitolo, nella sezione pc, abbiamo visto che era possibile definire una propria callback (la idle() function) di animazione separatamente dalla callback di disegno del singolo frame (la display() function) In OpenGL ES, invece, manca questa divisione. Se, ad esempio, si vuole disegnare un triangolo che ruota ad una certa velocità, un modo per farlo è memorizzare in una variabile il tempo in cui è stato effettuato il disegno del frame precedente, fare la differenza con il tempo di disegno del frame attuale, ottenendo così l' intervallo trascorso tra il frame precedente e quello attuale, calcolare l' effettiva rotazione e disegnare il triangolo ruotato di questo angolo. Niente di particolarmente difficile, ma sicuramente poco comodo, soprattutto quando ci sono molteplici oggetti in scena.

Tutto qui? Più o meno. Da aggiungere c' è il fatto che se invece si desidera lavorare con OpenGL ES 2.0 è necessario dichiararlo nel manifest xml dell' applicazione, pena il rischio di una terminazione forzata da parte del sistema operativo. L' altro aspetto da segnalare è che, trattandosi di un binding delle librerie di più basso livello, per poter specificare coordinate di vertici, texture, normali e anche per poter passare certi parametri al sistema è necessario in molti casi usare dei buffer nio (dei buffer di byte strutturati e ordinati non secondo le specifiche della DVM ma in base ai dettami della parte nativa sottostante) in quanto non è possibile passarli direttamente come array java.

4.3 Uno sguardo al mondo delle consolle di gioco

Analizziamo ora il settore delle consolle di gioco e notiamo che il mercato è dominato da tre grandi nomi:

- PlayStation 3, il prodotto di casa Sony, leader del settore, uscito in Giappone nel dicembre 1994 e arrivato in Europa nel settembre dell' anno successivo, giunto alla seconda incarnazione nel 2000 e uscito nel periodo 2006/2007 nella versione attualmente disponibile, la terza.

- Xbox 360 è il prodotto di casa Microsoft, diretto concorrente di PS3, uscito nella prima versione (Xbox) tra il 2001 e il 2002 e aggiornato alla versione attuale 360 a fine 2005.

- Nintendo Wii è il terzo concorrente del settore delle consolle, prodotto di casa Nintendo che è riuscito in poco tempo a conquistare un vasto pubblico grazie al suo innovativo sistema di controllo che puntava a coinvolgere fisicamente il giocatore.

- Assieme a PlayStation 3, Sony ha piazzato sul mercato anche la versione ridotta, la consolle di gioco portatile PlayStation Portable, uscita nel periodo 2004/2005 e giunta all' ultima versione, PlayStation Vita a febbraio 2012.

- Il Nintendo DS, la consolle portatile di casa Nintendo uscita tra il 2004 e il 2005, erede spirituale del GameBoy e GameBoy Advance è il diretto concorrente di PlayStation Portable e PlayStation Vita.

Chiunque decida di cimentarsi nello sviluppo di software per consolle di gioco deve prepararsi ad affrontare barriere e limitazioni di gran lunga maggiori rispetto a quelle accennate nella sezione precedente riguardanti lo sviluppo di app per i-Phone: a meno di non stipulare un oneroso contratto con la casa madre, è praticamente impossibile disporre di un ambiente di sviluppo per la consolle desiderata. L' unica eccezione è rappresentata da Xbox: Microsoft ha infatti studiato la

mossa di mettere a disposizione gratuitamente l' ambiente Microsoft XNA con l' intento di attirare gli sviluppatori software e combattere lo strapotere di mercato di PlayStation 3. Il punto di forza di XNA, oltre alla disponibilità a costo zero, è la possibilità di sviluppare software per Windows, per Xbox e per Windows Mobile, potendo convertire i progetti molto velocemente da un ambiente all' altro. Si Basa su DirectX 9 e .NET Framework, con l'utilizzo dell'IDE Visual Studio. Un' altra caratteristica molto apprezzata di XNA è la sua possibilità di poter lavorare sia ad alto livello che a basso livello, a discrezione dello sviluppatore. Ad esempio, è possibile applicare velocemente uno shader preconfigurato tra i molti già disponibili, ad esempio un effetto di luce ed ombra, come invece è possibile andare a scrivere personalmente uno shader o una struttura dati su cui operare (ad esempio la struttura della rappresentazione di un vertice), scegliendo ad esempio quanti bit allocare e come utilizzarli all'interno della memoria video. Per evidenti ragioni commerciali XNA supporta solo DirectX, che è la libreria grafica di riferimento del mondo Microsoft, quindi anche di Xbox.

Nintendo sfrutta invece un' api grafica proprietaria e, tanto Nintendo, quanto Sony, hanno blindato i loro ambienti di sviluppo ufficiali, che possono essere acquistati a caro prezzo solo dai professionisti che sono stati riconosciuti e approvati dalla casa madre.

Come già accennato nel capitolo 3, OpenGL ES 1.0 è l' api 3D ufficiale di PlayStation3 (anche se PS3 supporta alcune caratteristiche di OpenGL ES 2.0). Recentemente sono trapelate alcune release dell' SDK di Sony per PS3, tuttavia si tratta di leak pirata, di fatto quel software è stato rubato, quindi, anche se è possibile reperirlo in rete tramite siti pirata (con tutti i rischi e i potenziali problemi che questo può dare) è illegale farlo. In realtà è possibile reperire un SDK per PS3 legalmente e senza sborsare un solo centesimo: stiamo parlando del software "psl1ght", un homebrew sviluppato da una piccola ma esperta comunità di appassionati, di fatto si tratta di una versione modificata di GCC, distribuita insieme a delle librerie open source non ufficiali di terze parti prodotte tramite reverse engineering.

Strutturalmente psl1ght si basa su GCC, non potendo utilizzare il compilatore ufficiale. Su GCC poggia ps3toolchain, un compilatore e un insieme di tool che si usano nella creazione dell' applicazione

(più o meno in maniera simile a come Android usa il tool ANT). Tra le varie cose, ps3toolchain fa uso della gmplib, ossia la libreria GNU Multiple Precision Arithmetic Library. Sopra ps3toolchain sono definite ulteriori librerie, come le ps3libraries, che possono essere richieste o meno a seconda del tipo di applicazione. psl1ght fa uso anche del Nvidia Cg Toolkit, necessario per gli shader. Le parti più importanti che costituiscono psl1ght sono la RSXGL, un wrapping di OpenGL con cui fare grafica tridimensionale sfruttando l' hardware e SDL, per i contenuti multimediali quali suoni, filmati, eventi da periferiche, ma permette una certa integrazione anche con GLUT.

La compilazione della propria applicazione avviene con g++, mentre il linking e la produzione dell' eseguibile può essere automatizzata attraverso l' uso del "make pkg". A conti fatti, psl1ght è poco più di un compilatore e assemblatore, tuttavia sarebbe impensabile pretendere di avere a disposizione un emulatore per eseguire il debug delle applicazioni in quanto è impensabile riuscire ad emulare tutto l' hardware di una PS3 su un solo pc: il solo processore è costituito da 8 core, di cui sei vengono impiegati per il gioco, uno per la sicurezza e l' ottavo al momento non viene sfruttato, ciascuno di questi core può superare i 3Ghz. Un attuale pc domestico monta un dual core o nei casi migliori un quad core che a sì e no riesce a raggiungere i 3 Ghz. Quindi il modo più diretto per testare un' applicazione è copiarla su una memoria USB e lanciarla direttamente dall' interfaccia della PS3: mostrerà una cartella relativa all' USB e da questa sarà possibile eseguire l' applicazione, direttamente su PS3.

Nel nostro caso, si è deciso di muovere i primi passi nell' ambiente PlayStation 3 seguendo alcuni tutorial, il più completo dei quali è quello messo a disposizione da "lazy foo" (lazyfoo.net). Come prima prova si è deciso di scaricare il primo esempio di lazy foo, il classico "hello world". Dopo aver scaricato, dato un' occhiata e compilato il codice – il linguaggio di riferimento è il C++ – il compilato è stato copiato su un pendrive usb e avviato dalla schermata principale della XMB (XrossMediaBar, è l' interfaccia grafica della PS3, il nome è dovuto alla disposizione a croce degli elementi) di una PlayStation 3. Il risultato è stato positivo. PSXBrew (psxbrew.net) mette a disposizione un altro tutorial, più breve ma più diretto, in cui illustra le funzionalità fondamentali. Si è provato (con successo) l' esempio

del cubo texturizzato e si è cercato di provare a modificarlo per includere la gestione del controller, con cui ruotare tale cubo, tuttavia questa volta l' esito è stato fallimentare. Va anche detto che i due tutorial hanno livelli di accessibilità differenti, quello di lazy foo, benchè sia comunque per esperti, è più completo e maggiormente dettagliato, mentre psxbrew è veramente per intenditori. In entrambi i casi – e soprattutto nel secondo – ci troviamo alle prese con due prodotti del tutto inadatti ai principianti dell' informatica e alquanto difficili anche per degli informatici esperti che si avvicinano per la prima volta al mondo PS3. Probabilmente l' esito negativo con il secondo esperimento è dovuto anche a questo motivo, oltre alla ristrettezza dei tempi, tuttavia il primo successo fa ben sperare. In ogni caso l' obiettivo non sarebbe stato quello di effettuare un porting completo di "PunchingBalls Multiplayer" anche per PS3, ma solo quello di provare degli esempi con le funzionalità necessarie per il gioco (disegno tridimensionale, scritte a video, animazione e reazione ai comandi via controller, ecc). A quel punto il porting completo di "PunchingBalls Multiplayer" sarebbe stato soltanto un esercizio di programmazione, un' altra implementazione, questa volta in C++, di quanto discusso nel capitolo 3.

Capitolo 5

Conclusioni e possibili sviluppi

Per questa tesi si è scelto di realizzare un videogioco multiplayer online strutturato in una maniera molto simile a come lo sono i prodotti analoghi delle videogame house professionali, ovvero costruendo e dedicando una parte al server di gioco e distribuendo ai giocatori il client. In questo caso specifico ci si è poi prefissati l' obiettivo ulteriore di costruire il client non solo per pc, ma anche per piattaforme di gioco diverse dal pc. Più in dettaglio, dopo una progettazione il più possibile universale, il client è stato implementato sia su pc Linux che su smartphone Android.

Dopo una prima fase di studio degli aspetti di un videogioco, sia come prodotto finale, sia sotto il profilo tecnico, si è inizialmente realizzato un sistema minimale che facesse da prototipo, nel quale molti dei contenuti dell' applicazione finale non erano ancora presenti. Interessava in questa fase valutare l' effettiva fattibilità, sia dal punto di vista della distribuzione, sia dal punto di vista dell' usabilità da parte degli utenti. Appurato che la cosa era fattibile e con buoni risultati, sono stati aggiunti tutti i dettagli (e le schermate necessarie) al client pc per renderlo completo e al server per gestire molti client contemporaneamente.

Una prima prova di test è stata effettuata subito dopo aver completato il client per pc: su un sistema con processore Q6600, 2 GB di ram e Windows XP sono state virtualizzate due macchine con Ubuntu

9.10, e su ciascuna di esse era in esecuzione un client per pc. Sullo stesso sistema è stato fatto girare anche il server. Nonostante tali condizioni, il gioco ha dimostrato una fluidità più che accettabile.

Successivamente, dopo una fase di studio e valutazione del sistema Android, e stato implementato il client per smartphone. Una volta completato il client per Android, è stato effettuato un test in condizioni simili al caso precedente, sullo stesso sistema fisico, dove al posto di un Ubuntu virtuale era in esecuzione un emulatore di Android. Sebbene un po' "gommoso" sull' emulatore di Android, la giocabilità era ad un livello quasi accettabile.

In ogni caso l' emulatore di Android non è certo il posto più adatto per testare un' applicazione come un videogioco. Per valutare l' effettiva fluidità del client per smartphone test più seri sono stati svolti su un vero dispositivo, un Samsung Galaxy Ace. Il risultato è stato decisamente molto più accettabile e soddisfacente.

Più leggero è stato il lavoro svolto nell' ambiente PlayStation 3, si è trattato di analizzare la situazione e sperimentare una prova di fattibilità. Nonostante l' accessibilità quasi nulla e la ristrettezza dei tempi, quei pochi buoni risultati ottenuti lasciano ben sperare.

In definitiva l' obiettivo di sviluppare un videogioco multiplayer online è stato raggiunto con il completamento del server e del client per pc. L' obiettivo di sviluppare il gioco (e in particolare il client del gioco) anche su sistemi diversi dal pc è stato raggiunto durante la fase "Android" della tesi. Entrambi gli obiettivi hanno infine un punto di contatto nel senso che un' applicazione fatta in questo modo dovrebbe permettere di mettere in comunicazione (e in questo caso in competizione) i giocatori indipendentemente dalla specifica piattaforma sulla quale è in esecuzione il gioco. E anche questo è stato soddisfatto: giocatori su pc possono disputare match con giocatori su smartphone e viceversa.

Un possibile sviluppo futuro, anche a breve termine, è l' integrazione della parte single-player già sviluppata in passato con la parte multi-player sviluppata in questa tesi e la commercializzazione di un simile prodotto.

Un altro possibile scenario è il perfezionamento dello strumento usato per costruire il gioco, il generatore di codice citato nel capitolo 3, tra le altre cose includendo il porting sotto molteplici piattaforme

e ambienti software. Uno strumento che in automatico e in pochi secondi riesce a costruire tutta l' infrastruttura e buona parte del codice finale permette un notevole risparmio sui tempi di sviluppo. Se poi permette di fare queste cose anche in molteplici ambienti è un valore aggiunto per gli sviluppatori, che riuscirebbero in poco tempo a portare il proprio prodotto su più mercati.

Ringraziamenti

Chiunque sia riuscito a sopportarmi in questo periodo merita questa sezione.

Dai miei genitori, che mi hanno sempre sostenuto e incoraggiato in ogni situazione, ai miei amici, sia coloro con i quali ho condiviso questo cammino, sia coloro che mi hanno visto... anzi, che in realtà non mi hanno più visto, e per finire ringrazio anche il mio relatore, che mi ha dedicato parte del suo tempo in questo lavoro e ha dimostrato di avere davvero molta pazienza per riuscire a seguirmi e consigliarmi.

Bibliografia

Si riportano le fonti più importanti:

- OpenGL e OpenGL ES:

  http://www.opengl.org

  http://www.khronos.org/opengl

  http://www.khronos.org/opengles

  Beginning OpenGL Game Programming, Dave Astle, Kevin Hawkins

  OpenGL Reference Manual: The Official Reference Document to OpenGL, Version 1.4 (4th Edition), Addison-Wesley Professional, 2004.

  OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 2 (5th Edition). Addison-Wesley Professional, 2005.

- Informazioni, materiale e approfondimenti Android:

  http://developer.android.com

- Informazioni e materiale PSL1GHT:

  http://psl1ght.com

  http://psl1ght.net